TRS-80® MODEL III

FORTRAN

Catalog Number

26-2200

Radio Shack TRS-80 SOFTWARE™

CUSTOM MANUFACTURED IN USA BY RADIO SHACK, A DIVISION OF TANDY CORP.

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY; LOSS OR DAMAGE CAUSED DR ALLEGED TD BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

Model 4 FORTRAN Program:
©1983, Microsoft, Inc.
Licensed to Tandy Corporation
All Rights Reserved.

All portions of this software are copyrighted and are the proprietary and trade secret information of Tandy Corporation and/or its licensor. Use, reproduction or publication of any portion of this material without the prior written authorization by Tandy Corporation is strictly prohibited.

TRSDOS 6 Operating System:
©1983, Logical Systems, Inc.
Licensed to Tandy Corporation
All Rights Reserved.

All portions of this software are copyrighted and are the proprietary and trade secret information of Tandy Corporation and/or its licensor. Use, reproduction or publication of any portion of this material without the prior written authorization by Tandy Corporation is strictly prohibited.

Model 4 FORTRAN Manual
©1984, Tandy Corporation
All Rights Reserved.

Reproduction or use, without express written permission from Tandy Corporation and/or its licensor, of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors or omissions in this manual, or from the use of the information contained herein.

1Ø 9 8 7 6 5 4 3 2 1

An Overview of the MODEL 4 Fortran Package

This manual describes how to use your FORTRAN package (Cat. No. 26-2219) with the TRS-8Ø operating system (TRSDOS).

We assume you know how to program and are familiar with other computer languages, such as BASIC.

Your FORTRAN manual is in two main sections:

- Operating FORTRAN, an introductory section that describes the systems in the package and how they interrelate.
- The FORTRAN Language, which discusses the statements and functions of the language.

Your Model 4 FORTRAN package includes four systems:

- **The Editor,** ALEDIT/CMD, for writing and editing FORTRAN source files on diskette. You can also use it to edit BASIC programs.
- **The Compiler,** F8Ø/CMD, which reads your FORTRAN source program from diskette, translates it into relocatable object code, and saves it on diskette.
- **The Linker,** L8Ø/CMD, which lets you input the relocatable object program, execute it (by linking it with all the subroutines it has called), and save it on your diskette as a TRSDOS command file. Your resulting TRSDOS command file is a complete, independent object program that you can load and execute easily and quickly from TRSDOS.
- **The Assembler,** M8Ø/CMD, which lets you create relocatable assembly language subroutines that you can link to your FORTRAN programs.

These systems are all contained on one diskette.

Important Note: Be sure to make backup copies of your FORTRAN diskette before proceeding. See your disk system's owner's manual if you need instructions on how to do this.

This manual assists you in learning TRS-8Ø FORTRAN, but it is not a tutorial on FORTRAN programming.

If you are new to FORTRAN and need help learning the language, we suggest the following books:

1. Microsoft FORTRAN by Paul Chirlian (dilithium, 1981)

2. Guide to FORTRAN-IV Programming by Daniel McCracken (Wiley, 1965)

3. Ten Statement FORTRAN Plus FORTRAN IV by Michael Kennedy and Martin B. Solomon (Prentice-Hall, 1975, Second Edition)

4. FORTRAN by Kenneth P. Seidel (Goodyear, 1972)

5. FORTRAN IV, A Self-Teaching Guide by Jehosua Friedmann, Philip Greenberg, and Alan Hoffbert (Wiley, 1975)

6. FORTRAN, A Structured, Disciplined Style by Gordon B. Davis and Thomas R. Hoffman (McGraw-Hill, 1978)

NOTATIONS

In this manual, we use the following notations:

ALL CAPS
commands, statements, or functions. Type these exactly as shown.

underline
values that you must supply.

CONTENTS

Part 1 -- Operating FORTRAN 9
Chapter 1 Introduction 11
Chapter 2 The Editor (ALEDIT) 23
Chapter 3 The Compiler (F8Ø) 39
Chapter 4 The Linker (L8Ø) 49

Part 2 -- The FORTRAN Language 57
Chapter 5 FORTRAN Program Structure 61
Chapter 6 Data 75
Chapter 7 Expressions 87
Chapter 8 Input/Output 95
Chapter 9 Segmenting Programs 121
Chapter 1Ø FORTRAN Statements 135
Chapter 11 FORTRAN Functions 187

Part 3 -- Error Messages 213
A/ TRSDOS Errors 215
B/ Compiler Error Messages 217
C/ Editor Error Messages 222
D/ Linker Error Messages 224

Part 4 -- Quick Reference 227
A/ Editor 229
B/ Compiler 234
C/ Linker 236
D/ FORTRAN Statements and Functions 237

Appendices 245
A/ Language Extensions and Restrictions 247
B/ I/O Interface 249
C/ Subprogram Linkages 254
D/ ASCII Codes 257
E/ FORLIB Arithmetic Library Subroutines 258
F/ Output File Format 261
G/ Storage Format 262
H/ Format of Link-Compatible Object Files 264
I/ MACRO-8Ø Assembler 267

Index 311

Section I

Operating FORTRAN

Part 1

OPERATING FORTRAN

CHAPTER 1 / INTRODUCTION

To see how to use your FORTRAN package, go ahead and run a program.

To do this, follow these steps:

(1) Type a FORTRAN program
(2) Check the program for errors (by compiling it)
(3) Save the compiled program
(4) Save the program as a TRSDOS command file
(5) Execute the program

Step 1. Typing a FORTRAN Program

Insert your FORTRAN diskette and press the reset button of your computer. Type in the date and time. Then your screen displays:

```
TRSDOS Ready
```

Type:

```
ALEDIT <ENTER>
```

This loads the Editor. The Editor lets you write and edit your FORTRAN programs.

At the "Ready" prompt, type I (the ALEDIT command for Insert):

```
I
```

This command clears the screen and displays the name of the file to be edited at the upper right corner of the screen.

Since no file was loaded with ALEDIT, the Editor assigns NONAME/SRC as the filename of the current insertion.

Use <right arrow> to space over to Column 9, and then type:

```
A = 12.5 <ENTER>
```

You may type FORTRAN statements anywhere between Column 7 and Column 72. Columns 1-5 are reserved for statement labels, and Column 6 is reserved for continuation markers.

Type the remaining lines of the program. Be sure to begin each line at Column 9. (Type the statement label in the format line in any of the first five columns.)

```
     WRITE(5,5) A <ENTER>
5    FORMAT(' A = ',F4.1) <ENTER>
     END <ENTER>
<BREAK>
```

When you type <BREAK>, your screen displays ^Brk^. This means the Editor is out of Insert Mode and ready to accept additional commands. (Remember, the first one it accepted was the Insert command.)

If you made a mistake while typing a line, press:

```
Q <ENTER>
```

This is the Quit command. It exits the Editor and returns you to TRSDOS Ready. Now repeat Step 1. In Chapter 2, we discuss how to edit each line, but for now you must retype the program.

When you are sure that the program is correctly typed, press <BREAK> to exit Insert Mode. Type:

```
W SAMPLE/FOR
```

This command writes your FORTRAN "source program" to the disk with a filename of SAMPLE/FOR. If you don't save an edited file with a W command, you lose your corrections when exiting

the Editor. The SAMPLE/FOR filename now appears at the upper right corner of your screen. If you save a program without a filename after the W, the Editor saves the file with the current filename. No warning is given when a file with the same name is overwritten with the W command.

If you don't assign an extension to a filename, the Editor automatically attaches the SRC extension.

You are now finished with the Editor. Exit it and return to TRSDOS Ready by typing:

Q <ENTER>

Step 2. Check the Program For Errors

Each time you run a program, check for syntax errors, which are usually simple spelling and word order mistakes in your statement lines. You can perform the check by compiling the program. Type:

F8Ø <ENTER>

F8Ø is the filename of the Compiler. The computer loads the Compiler and then displays the asterisk (*) prompt. To perform the syntax check, type:

=SAMPLE/FOR <ENTER>

This tells the Compiler which file to compile. Your FORTRAN Compiler then checks the syntax of your program. (You may omit /FOR. The Compiler uses /FOR as a default extension on your source files.)

During processing, your screen displays $MAIN, along with any syntax errors. Some sample errors are:

?Line: ØØ3ØØ Statement Is Out Of Sequence:Format'

?1 Fatal Error(s) Detected

If you have errors, return to TRSDOS by pressing <BREAK>. Then type:

REMOVE SAMPLE/FOR <ENTER>

and repeat Steps 1 and 2 again. (The Error Messages are listed in Part 3.) When the program compiles successfully, you can save it on diskette as a relocatable object file.

Step 3. Create a Relocatable Object File

Before executing your FORTRAN source program, you must save it on your diskette as a "relocatable" object file. A relocatable object file is your source program, compiled into object code. The file is "relocatable" since it can be located in different parts of the memory.

You create this file with the Compiler. You should be in the Compiler at this point, with your screen displaying the asterisk prompt. If you are in the TRSDOS Ready mode instead, type:

F8Ø <ENTER>

to reenter the Compiler. To get ready to execute your program, type:

BOAT=SAMPLE <ENTER>

If you would also like technical information about the relocatable addresses of BOAT, type this instead:

BOAT,CAR=SAMPLE <ENTER>

The computer compiles your source file, SAMPLE/FOR. (Since you did not specify an extension for this source file, the Compiler assumes the extension is /FOR.)

It saves the compiled file on diskette as a relocatable object file named BOAT. If you chose the second option, the computer also creates a "listing" file, named CAR, on your diskette.

You can see all these files on your diskette's directory. Press <BREAK> to return to TRSDOS Ready. Now type:

DIR <ENTER>

Notice that the computer added the extension /REL to BOAT, the "relocatable" file, and /LST to CAR, the "listing" file. If you did not create "CAR," now go to Step 4.

CAR/LST contains useful technical information about BOAT/REL. You can list it by typing (in response to the TRSDOS Ready prompt):

LIST CAR/LST <ENTER>

Your screen displays:

```
FORTRAN-80 VER. X.XX Copyright 1978-1981 (C) By Microsoft
BYTES: 18287
CREATED: XX-XXX-XX

1                                       A=12.5
*****     0000'     LD        BC,$$L
*****     0003'     JP        $INIT
2                                       WRITE(5,5) A
*****     0006'     LD        HL,[00 00 48 84]
*****     0009'     CALL      $L1
*****     000C'     LD        HL,A
*****     000F'     CALL      $T1
*****     0012'     LD        DE,5L
*****     0015'     LD        HL,[05 00 00 00]
*****     0018'     CALL      $W2
3         5                             FORMAT(' A = ',F4.1)
*****     001B'     LD        DE,A
```

```
*****     ØØ1E'     LD        HL[Ø1 ØØ ØØ ØØ]
*****     ØØ21'     LD        A,Ø2
*****     ØØ23'     CALL      $I1
*****     ØØ26'     CALL      $ND
4                               END
*****     ØØ29'     CALL      $EX
*****     ØØ2C'     Ø1 ØØ ØØ ØØ
*****     ØØ3Ø'     ØØ ØØ 48 84
*****     ØØ34'     Ø5 ØØ ØØ ØØ

Program Unit Length=ØØ38 (56) Bytes
Data Area Length=ØØ13 (19) Bytes

Subroutines Referenced:

$I1       $INIT     $L1

$T1       $W2       $ND

$EX

Variables:

A         ØØØ1"

Labels:

$$L       ØØØ6'     5L   ØØØ5"

1
```

Note: Some of these memory location numbers may vary slightly.

You can halt the listing by pressing <SHIFT> <@>. Press any key to continue.

You see the following items in the listing file:

(A) the number of bytes remaining in memory.
(B) the actual FORTRAN statements from your source file.

(C) the relocatable object-code instructions from the relocatable object file. The Compiler has disassembled these instructions into Z-8Ø assembly language code to help you read the code. Notice that each instruction is located in a relocatable address. For example, the instruction:

```
JP   $INIT
```

resides at the address of ØØØ3' (hexadecimal). When loaded, this instruction is located at ØØØ3 plus the originating address of the program. Also notice that most of these instructions are calls or jumps to subroutines. For example:

```
JP   $INIT
```

jumps to the subroutine $INIT.

```
CALL $L1
```

jumps to the subroutine $L1.
These subroutines are all on your diskette in a file named FORLIB/REL. When you execute the program, use the Linker to load FORLIB into memory and look up the address of each instruction.

(D) the physical length of the program (in bytes). 38 is the length in hexidecimal notation; 56 is the length in decimal.

(E) the physical length of the data in the program. 12 is the hexidecimal length; 18 is the decimal length.

(F) all the subroutines required to execute the program.

(G) the relocatable addresses where the value of your variables reside. In this case, there is only one variable, A, which contains the value of 12.5. It resides at the relocatable area of ØØØ1". When loaded, 12.5 resides at ØØØ1 plus the originating address of the data in your program.

(H) the relocatable addresses that the labels in your

program reference, or "point to." When loaded, $$L points to ØØØ6 plus the originating address of the program; 5L points to ØØØ5 plus the originating address of the data. Notice that the single quote (') signifies a relocatable program address. Double quotes (") signify a relocatable data address.

To print out a listing of this file, type:

```
LIST CAR/LST (P) <ENTER>
```

Now you are ready to execute the program.

Step 4. Saving Your Program as a Command File

Before executing your program you must save your object file as a TRSDOS command file.

To do this, you need to have BOAT in the Linker. If you are in TRSDOS Ready, type:

```
L8Ø <ENTER>
```

Then type:

```
BOAT <ENTER>
```

Your screen displays:

```
DATA 3ØØØ 3Ø4B <   75>

FORLIB RQUEST
-$EX      3Ø3D      -$I1      3Ø37      -$INIT    3Ø17
-$L1      3Ø1D      -$ND      3Ø3A      -T1       3Ø23
-$W2      3Ø2C
     7 UNDEFINED GLOBAL(S)
     43427 BYTES FREE
```

Note: These numbers may vary slightly.

The seven "undefined globals" are the seven subroutine requests needed to link and run the simple program of A=12.5. The characters on the left are the names of the subroutines (-$I1, -ND, and so on). The subroutines are located in the FORLIB subroutine library. The numerals on the right are the memory addresses of your program that contain the requests for these subroutines (3Ø37, 3Ø3A, and so on).

The globals are undefined because the Linker has not yet input FORLIB into memory and looked up where they reside in memory.

You can then save the object file by writing the name of the file (you have to assign a new name), followed by a -N and a -E.

Assign the name "FAN" for your command file. Type:

FAN-N-E <ENTER>

Now the program is stored in a file called FAN/CMD. (The computer gave it the extension /CMD to signify that it is a TRSDOS command file.)

FAN/CMD is an independent object file that you can run directly from TRSDOS Ready. When you want to execute it, type:

FAN <ENTER>

Step 5. Execute the Program

To execute the program, type the name of the file. Type:

FAN <ENTER>

This command tells TRSDOS to load and run the program FAN.

While the program is executing, you may get a "Compiler runtime error message." This message appears on your screen as two letters enclosed with asterisks. If you get this message, look up the meaning in Part 3, REMOVE SAMPLE/FOR, and go back to Step 1.

The sample program runs quickly because it is simple. The longer and more complicated your FORTRAN programs are, the longer they take to run and link. This is because a more complicated program requires more subroutine library requests.

Review

When you wrote, compiled, linked, and executed the preceding program (SAMPLE/FOR), you followed this procedure:

(1) First you wrote the FORTRAN program using the Editor (ALEDIT). You saved this on diskette as a source file.

(2) Then you checked the program for correct syntax using the Compiler (F8Ø).

(3) Next you created the relocatable object file (BOAT/REL) that contains the object code. You might have also created a listing file (CAR/LST). Creating a listing file is optional.

(4) Then you created a command file (FAN/CMD) so that you can run the program directly from TRSDOS.

(5) Finally you executed the program.

Here are some shortcuts you can take once you understand the procedure:

(1) Write the program in the same way you did above.

(2) Check the program for the correct syntax by typing (in TRSDOS Ready):

F8Ø =SAMPLE <ENTER>

(3) Create the relocatable object file by typing (in TRSDOS Ready):

F8Ø BOAT=SAMPLE <ENTER>

(4) Load the object file and create a command file at the same time by typing (in TRSDOS Ready):

L8Ø FAN-N,BOAT-E <ENTER>

(5) Execute the program by typing the name of the command file (in TRSDOS Ready):

FAN <ENTER>

If you think your program is error-free, you can omit Step 2. Step 3 reports syntax errors as it creates the relocatable object file.

CHAPTER 2/ THE EDITOR (ALEDIT)

ALEDIT lets you enter and edit FORTRAN source programs. You can save these programs on disk as source files to be compiled into object code.

This selection describes how to use ALEDIT. For information on how to write a FORTRAN language source program, see Part 2, "The FORTRAN Language."

> **Note:** Since the FORTRAN Compiler's ALEDIT Editor also serves as the Editor for the ALDS Assembly Language Development System (Radio Shack Cat. No. 26-2Ø12), some Insert Mode control codes generate special assembly language characters. These characters have no function in this FORTRAN Compiler program, and therefore are not listed in this manual.

LOADING THE EDITOR

To load ALEDIT and the specified source filespec, at the TRSDOS Ready prompt type:

ALEDIT source filespec

The source filespec is optional. For example:

ALEDIT <ENTER>

also loads the Editor. Your screen displays the following heading, which is similar to that displayed when you specify source filespec:

TRS-8Ø Model 4 Text Editor Version v.r.p.
Copyright (c) 1982, 83 Tandy Corp.

(v.r.p. is the version, release, and patch numbers.)

If you load ALEDIT without a filespec, the screen displays a "Ready" prompt below the copyright message. To enter Command Mode and begin writing your FORTRAN program, type:

```
I
```

(Do not type <ENTER>.) However, typing

```
ALEDIT SAMPLE/FOR <ENTER>
```

loads the Editor, displays the above heading, and then loads a source file named SAMPLE/FOR.

If the source filespec does not contain an extension, the Editor appends /SRC to it.

ALEDIT loads into all the memory above TRSDOS. It reserves approximately the top 40K bytes in the Model 4 as an "edit buffer" for inserting your programs. However, if you also load one of the High Memory TRSDOS utilities, the edit buffer is smaller.

USING THE EDITOR

You can use ALEDIT in the Insert mode, Command Mode, and the Line Edit Mode.

THE INSERT MODE

Pressing I places the Editor in the Insert mode and allows you to enter programs and text. If you make an error while typing in a program line, but haven't pressed <ENTER> yet, you may use the right and left arrow keys to move the cursor and immediately correct the mistake. If, however, you have already pressed <ENTER>, you may only make changes by returning to the Command

Mode. Pressing <BREAK> exits the Insert mode and returns you to the ALEDIT Mode.

THE COMMAND MODE

When you first load the Editor, it is in Command Mode. While in this mode, you can use any special keys listed in Table 1 or the commands listed in Table 2.

All commands except I (Insert Mode) and E (Line Edit Mode) return to Command Mode after executing. To return to Command Mode from I, press <BREAK>; to return to Command Mode from E, press <ENTER>.

An ALEDIT command creates a blank "work line" and points to the line just beneath it. To redisplay the screen after an error message and delete the work line, use the N command.

Sample Use

Use the I command to insert this program:

```
THIS IS THE FIRST LINE <ENTER>
THIS IS THE SECOND <ENTER>
AND HERE IS ANOTHER <ENTER>
AND ANOTHER <ENTER>
<right arrow> END <ENTER>
```

Press <BREAK> to return to Command Mode.

You can move the cursor and rearrange the lines of the program. For example, type:

```
T
```

The cursor moves to the top of the text. Type <B> to move it to the bottom. Press <up arrow> and <down arrow> to move it to specific lines.

Move the cursor to Line 3 and type:

```
1
```

The less than symbol (<) appears to the left of the line. This specifies the beginning of a block. Move the cursor to line 4 and type:

```
2
```

The greater than symbol (>) appears to the left of the line. This specifies the last line in the block. Move the cursor up to Line 2 and type:

```
O
```

This command copies the block between Lines 1 and 2. Move the cursor to the next to last line and type:

```
D
```

This is the delete command, which executes without pressing <ENTER>. It deletes the last line.

To save this program on disk, type:

```
W TEST <ENTER>
```

It does not matter at which line the cursor is positioned. This saves this program on disk as a file named TEST/SRC. (**Note:** The W filespec command overwrites files with the same name without a FILE ALREADY EXISTS error message.)

Now, to exit the Editor, type:

```
Q <ENTER>
```

Q exits the Editor without writing the text to disk. If you forgot to save the text first, type ALEDIT * <ENTER> to reenter the Editor. This retains your text.

Be sure you use the ALEDIT * command immediately after you exit the Editor. It does not work predictably after you run a command that modifies memory. Also, be sure you type a space between ALEDIT and the asterisk(*).

Table 1 / ALEDIT Command Mode Keys

Keys	Description
<left arrow>	Moves the cursor one position to the left.
<down arrow>	Moves the cursor down one line (ignored if the cursor is not in Column 1).
<up arrow>	Moves the cursor up one line (ignored if the cursor is not in Column 1).
<CTRL><A>	Moves the cursor to the top of the screen.
<CTRL><B>	Moves the cursor to the bottom of the screen or to the first line after the last line of text.
<.>	Displays the current line sequence number. This number changes as you insert and delete lines.
#line<ENTER>	Moves the cursor to the specified line sequence number and moves that line to the top of the screen.
<BREAK>	Cancels any command being executed and returns to Command Mode.
<SHIFT> <up arrow>	Cancels the current command line if you have not yet pressed <ENTER>.

Table 2/ ALEDIT Editor Commands

current line
line where the cursor is currently positioned.

del (delimiter)
one of the following characters, which marks the beginning and ending of a string:
! " # $ % & ' ( ) * + , - . / : ; < = > ?

string
one to 37 ASCII characters.

text
source program or text currently in RAM.

A <ENTER>
reexecutes the last executed command. This command only works with the Editor commands C, F, X, L, and W.

B
moves the cursor to the bottom of the text.

C del string1 del string2 del occurrence <ENTER>
changes string1 to string2 for the number of occurrences you specify. Occurrences must be in the range 1 to 255. The changes begin at the current line and are made only to the first occurrence on a given line.

If you omit occurrence, only the first occurrence of string1 is changed. You may specify occurrence with an asterisk, in which case the first occurrence of string1 changes in all the remaining lines.

For example:

C/TEST/FILE/3 <ENTER>

changes the first 3 occurrences of TEXT to FILE.

C?TEXT?FILE?* <ENTER>

changes all occurrences of TEXT to FILE. (Change acts on only the first occurrence within a line.) After executing the command, the cursor positions itself at the last change or at the top of the file if changes went through the whole file.

D
deletes the current line or block of lines. To delete a block, position the cursor at the first line in the block and type <1>. Then position it at the last line and type the D command. (The block may be on several pages.) You must position the cursor on a line within the file.

For example:

```
          A = 12.5
<1>       WRITE (5,5) A
<D>     5 FORMAT (' A =  ',F4.1)
          END
```

deletes all but the following:

```
A = 12.5
END
```

You an cancel a block deletion after pressing <1> but before typing D. To do this, press <3>.

E
lets you edit the current line using Line Edit Mode subcommands. The line appears in reverse video. See "The Line Edit Mode" for a listing of subcommands.

F del string del occurrence **<ENTER>**
finds the specified occurrence of string. If you omit occurrence, it finds the first occurrence of string. If you omit string, it finds the last string specified. Occurrences must be in the range 1 to 255. For example:

```
F/TEXT/2 <ENTER>
```

finds the second occurrence of TEXT.

```
F/TEXT/ <ENTER>
```

finds the next occurrence of TEXT.

```
F <ENTER>
```

finds the next occurrence of the last specified string.

```
F%     % <ENTER>
```

finds the next occurrence of five blank spaces. The Editor searches for only one occurrence of the string in each line.

G <ENTER>
deletes all text from the current line to the end. The Editor first prompts you with:

```
"Are you sure?"
```

Type Y <ENTER> to delete; N <ENTER> to cancel.

H <ENTER>
prints the entire text if entered as the first command or the specified block on the printer. To print a block, move the cursor to the first line of the block and type <1>. Move the cursor to the last line of the block and type <H>. For example:

```
<1>          A = 12.5
             WRITE (5,5) A
<H>        5 FORMAT (' A =   ',F4.1)
             END
```

prints a block of text that includes the first three lines.

You can cancel a block printing after pressing <1> but before typing H. To do this, press <3>.

Press <BREAK> to terminate printing. If the printer is offline or goes offline during printing, you may lose some characters.

I
enters Insert Mode for inserting lines just before the current line.

J
displays current size of text and how much memory remains. Memory size does not include a small work area when the buffer is full, but the text size may reflect some of this work area.

K <ENTER>
deletes ALL text. (Does not delete text from the disk file, only from the edit buffer). Before deleting your text, the Editor asks you "Are you sure?" Type Y <ENTER> to execute the command; N <ENTER> not to execute it.

L filespec **$C <ENTER>**
loads filespec into the Editor. $C is optional. If specified, the Editor chains the new filespec to the end of the text currently in memory. If not specified, the new filespec overlays the current text.

For example:

L TEST <ENTER>
loads TEST/SRC into the Editor.

L TEST $C <ENTER>
chains TEST/SRC to the end of the text currently in memory.

The Editor loads fixed length record (FLR) files with a record length of one. If the file is fixed length, you must end each line with a carriage return.
Note: When the Editor completes, the record length is 256.

M
moves the specified block just ahead of the current line. Use <1> and <2> to specify the block.
For example:

```
          <1>       CALL READER (RESULT,N)
          <2>       CALL DSKRIT (RESULT,N)
                    AVERAG = AVE(RESULT,N)
                    STDDEV = STD(RESULT,AVERAG,STDDEV)
          <M>       CALL WRITER (RESULT,N,AVERAG,STDDEV)
```

moves the block of CALL instructions just ahead of the last line:

```
                    AVERAG = AVE(RESULT,N)
                    STDDEV = STD(RESULT,AVERAG,STDDEV)
                    CALL READER (RESULT,N)
                    CALL DSKRIT (RESULT,N)
                    CALL WRITER (RESULT,N,AVERAG,STDDEV)
```

You can cancel the block after specifying it but before typing M. To do this, press <3>.

N
updates the display. You might want to use this after executing the J command or canceling the G command.

O
copies the specified block just above the current line. (Use <1> and <2> to specify a block as described in the M command.)

P
moves the cursor to the next page (which is 24 lines from the top of the screen).

Q <ENTER>
exits the Editor. If you forgot to save the file first, type ALEDIT * <ENTER> immediately upon exiting the Editor. The Editor loads with your text retained in memory.

R <ENTER>
deletes the current line and enters Insert Mode. After using the J command, if there is ØØØØ memory left in the buffer, executing the R command deletes the line but does not let it be replaced with next text.

T
moves the cursor to the top of the text.

U
moves the cursor to the previous page (which is the 24 preceding lines).

V
scrolls current line to the top of the screen.

W filespec $optionl... <ENTER>
saves all text on disk as filespec. filespec is optional; if omitted, it is the filespec you used to load the file. The Editor appends /SRC to filespec unless it already includes an extension.

The options are:

E	Exits the Editor after saving the file unless there is an error.
L, ML, OR LM	Saves the file with line numbers in this format: ASCII line number/dummy TAB/text. **Note:** Do not save your source files with these options. The FORTRAN Compiler automatically assigns line numbers to source programs. The numbers, however, are visible only when you LIST the program.
M	Saves the file as a fixed length record (FLR) file with a LRL of 256 in this format: text/carriage return This option is the default. You can use ALEDIT to edit a "DO-file" created with the TRSDOS "BUILD" command and save this format, which the TRSDOS "DO" command can load.

For example:

```
W SAMPLE <ENTER>
```

saves all text as a file named SAMPLE/SRC.

```
W SAMPLE $E
```

saves text as SAMPLE/SRC. The Editor exits to TRSDOS Ready after saving the file.

Without using the L or the M options, the Editor saves the file in the format required by the FORTRAN Compiler

- Each character is saved exactly as it appears on the display.

- No carriage returns or end of text code is saved.
- Each line is saved in this format:
  length/text/

X del string1 del string2 del occurrence

Same as the C command, but prompts before making the change. Occurrence must be in the range 1 to 255.

The LINE EDIT MODE

The E command enters Line Edit Mode for editing characters within the current line.

Position the cursor on the line you wish to edit while you are still in the Command Mode. Enter Line Edit Mode by pressing E. When you enter this mode, the Editor displays the line in reverse video. You can then use any edit subcommands listed in Table 5 or the special edit keys listed in Table 6.

For example, assume the cursor is on the following line:

```
THIS IS THE FIRST LINE
```

To change the word FIRST to THIRD from the command mode, type:

```
E
```

(Do not press <ENTER>.) The Editor displays the line in reverse video. You are now in Line Edit Mode.

Use the <SPACEBAR> to position the cursor at the F in FIRST and type:

```
5CTHIRD <ENTER>
```

This stores the change and returns to Command Mode.

Table 3/ ALEDIT Line Edit Mode Subcommands

COMMAND	DESCRIPTION
A	Clears all changes and reenters Line Edit Mode for the current line.
nCstring	Changes the next n characters to the specified string. If you omit n, only one character is changed. (Press <SHIFT> <up arrow> to exit the change early.)
nD	Deletes n characters. If you omit n, one character is deleted.
E	Exits the Line Edit Mode and stores changes.
Hstring	Deletes the remaining characters, enters Insert mode, and lets you insert a string.
Istring	Lets you insert material beginning at the current cursor position on the line. Pressing <left arrow> deletes characters from the line. The line may be a maximum of 78 characters.
nKcharacter	Kills all characters preceding the nth occurrence of the character.* If you omit n, the first occurrence is used. If no match is found, the rest of the line is killed.
L	Moves the cursor to beginning of line.

Q	Quits the edit mode, canceling all changes.
nScharacter	Positions the cursor at the nth occurrence of character.* If no match is found, positions the cursor at the end of the line.
Xstring	Moves the cursor to the end of the line, enters Insert Mode, and lets you insert a string.

* The compare begins on the character following the current cursor position.

Table 4/ ALEDIT Line Edit Mode Special Keys

<SPACEBAR>	Moves cursor one position to the right.
<SHIFT> <up arrow>	Returns to Command mode from the I,X, C, or H subcommands.
<right arrow>	Moves the cursor to next tab position (or the end of the line) while in the I, X, or H subcommand mode.
<left arrow>	Moves cursor one position to the left.
<ENTER>	Identical to the E subcommand.

CHAPTER 3 / THE COMPILER

Your source file contains the FORTRAN statements you use to write your program. However, your computer cannot directly process the FORTRAN statements in the source file. The statements must be compiled.

Compilation is the process of creating object code from a source language. The computer understands this code directly.

The relocatable object code your Compiler creates consists of many calls to subroutines, which are contained in the FORLIB subroutine library.

To execute the compiled program, you save the relocatable object file and then use the Linker (discussed in Chapter 4) to link it to the subroutines and create an executable program.

Running the Compiler

To run the Compiler, type:

F8Ø <ENTER>

F8Ø (or more accurately, F8Ø/CMD) is the name of the Compiler file on your diskette. When the Compiler is ready to accept your commands, it prompts you with an asterisk. To exit the Compiler and return to TRSDOS (when you are at the command level), press <BREAK>.

COMMANDS

Your commands tell the Compiler the name of the source file you want to compile and what options you want to use. Here is the format for a Compiler command:

object filename,listing filename=source filename-switches

object filename -- This is the name you give your object file. It is optional. To create a relocatable object file, you must include this part of the command. The default extension for the object filename is /REL.

,listing filename -- This is the name you give the listing file. It is optional. The default extension for the listing file is /LST.
To send the listing to the printer, use LPT as the filename. To send the listing to the screen, use TTY.

=source filename -- This is the name of a FORTRAN program you have saved on diskette. The default extension for a FORTRAN source filename is /FOR. The source filename is always preceded by an equal sign in a Compiler command.

switches -- These affect the way the program is compiled. If you do not specify a switch, -H is used (default).

Note: All filenames must be in this standard filename format. (For further information, see Chapter 1.)

filename/extension.password:drive #

Examples

```
=SAMPLE/FOR
```

compiles the source file SAMPLE/FOR without creating an object file or listing file. This is the syntax test you can use on each source program file.

```
,SAMPLE.PASS=SAMPLE.PASS
```

compiles the source file SAMPLE/FOR.PASS and creates a listing file called SAMPLE/LST.PASS. (No object file is created.)

```
BOAT,CAR=SAMPLE
```

compiles the source file SAMPLE/FOR. Creates a relocatable object file called BOAT/REL and a listing file called CAR/LST. (This is exactly the same as the example in the Introduction.)

```
BOAT,CAR=SAMPLE-P-O
```

compiles the same files as above using the -P and -O switches.

```
TEST,TEST=TEST-N
```

compiles TEST/FOR and creates TEST/REL and TEST/LST using the -N switch.

SWITCHES

In programming, a switch is similar in concept to a household electrical switch (a device for turning something on or off).

When you give your computer instructions, it selects one of two paths -- on or off. You determine the path by setting the switch. You use the Compiler switches during compilation; each switch has a different effect on the program output.

In the FORTRAN Compiler, switches are always preceded by a hyphen (-). You may use more than one switch in the same command. The switches are:

-O	listing file addresses in octal
-H	listing file addresses in hexadecimal
-N	object code deleted from listing file
-P	allots extra stack space
-M	object code in ROM format

The -O, -H, and -N switches tell your Compiler what format to use in creating the listing file (the meaning of the listing file is explained in Chapter 1).

-H

Prints listing addresses in hexadecimal (the default condition). Even though this is a default, you must use this switch to get out of octal and back into hexadecimal. The listing file (CAR/LST) you compiled in the Introduction was hexadecimal.

-O

Prints listing addresses in octal.

Examples

```
BOAT,CAR=SAMPLE-O
```

(Type the letter O, not zero.)

compiles SAMPLE/FOR into the relocatable object file, CAR/LST. All memory addresses in CAR/LST are in octal notation. If you use this example to compile the SAMPLE/FOR file, which you created in Chapter 1, CAR/LST appears as follows:

```
1                                   A=12.5
*****     ØØØØØØ'   LD              DC,$$L
*****     ØØØØØ3'   JP              $INIT
2                                   WRITE(5,5) A
*****     ØØØØØ6'   LD              HL,[ØØØ ØØØ 11Ø 2Ø4]
*****     ØØØØ11'   CALL            $L1
*****     ØØØØ14'   LD              HL,A
*****     ØØØØ17'   CALL            $T1
*****     ØØØØ22'   LD              DE,5L
*****     ØØØØ25'   LD              HL,[ØØ5 ØØØ ØØØ ØØØ]
*****     ØØØØ3Ø'   CALL            $W2
3         5                         FORMAT(' A = ',F4.1)
*****     ØØØØ33'   LD              DE,A
*****     ØØØØ36'   LD              HL,[ØØ1 ØØØ ØØØ ØØØ]
*****     ØØØØ41'   LD              A,ØØ2
*****     ØØØØ43'   CALL            $I1
*****     ØØØØ46'   CALL            $ND
4                                   END
*****     ØØØØ51'   CALL            $EX
*****     ØØØØ54'   ØØ1 ØØØ ØØØ ØØØ
*****     ØØØØ6Ø'   ØØØ ØØØ 11Ø 2Ø4
*****     ØØØØ64'   ØØ5 ØØØ ØØØ ØØØ'

Program Unit Length=ØØØØ7Ø (56) Bytes
Data Area Length-ØØØØ23 (19) Bytes

Subroutines Referenced:

$I1       $INIT     $L1

$T1       $W2       $ND

$EX
```

```
Variables:

A          ØØØØØ1"

Labels:

$$L        ØØØØØ6'          5L         ØØØØØ5"

1
```

The listing addresses are the six-digit numbers that represent locations in memory for different calls (such as ØØØØØØ', ØØØØØ3', ØØØØØ6', and so on). Also, notice that all the addresses are in octal (no number higher than seven).

```
FISH,BIRD=ELEVEN-O
```

The program ELEVEN/FOR is compiled. A listing file called BIRD/LST and an object file called FISH/REL are created. The addresses in the BIRD/LST file are octal.

-N

The listing file lists only the FORTRAN source code and not the object code that is generated.

Example

```
BOAT,CAR=SAMPLE-N
```

compiles the program SAMPLE/FOR. An object file called BOAT/REL and a listing file called CAR/LST are created. CAR/LST contains only your FORTRAN source statements, not the corresponding (generated) object code.

If you use the example to compile the SAMPLE/FOR file from Chapter 1, CAR/LST appears as follows:

```
FORTRAN-8Ø VER. X.XX Copyright 1978-1981 (C) by Microsoft
BYTES: 27461
CREATED: XX-XX-XXXX

1               A=12.5
2               WRITE(5,5) A
3     5         FORMAT(' A = ',F4.1)
4               END

Program Unit Length=ØØ13 (56) BYTES
Data Area Length=ØØ13 (19) BYTES

Subroutines Referenced:

$I1             $INIT               $L1
$T1             $W2                 $ND
$EX

Variables:

A          ØØØ1"

Labels:

$$L        ØØØ6'          5L        ØØØ5"

1
```

In this listing file, only the source code (and not the object code) is listed. This is a convenient switch to use since you'll probably rarely want to look at the object code.

The remaining Compiler switches affect the following other aspects of compilation.

-P

For long programs, each -P switch allocates an extra 1ØØ bytes of stack space for use during compilation. If you get any "stack overflow" errors during compilation, use the -P switch.

Example

```
*BILL=STEVE-P-P
```

compiles the program STEVE/FOR and creates an object file called BILL/REL. Your Compiler is also allocated 2ØØ extra bytes of stack space (1ØØ extra bytes for each -P switch that you enter).

-M

Tells the Compiler that the object code must be in a form that can be loaded into ROMs. When you specify a -M, the generated code differs from normal in the following ways:

- FORMATs are in the program area, with a "JMP" around them.
- Parameter blocks (for subprogram calls with more than three parameters) are initialized at runtime, rather than being initialized by the loader.

Note: If you intend your FORTRAN program for ROM, be aware of the following:

1. Do not use DATA statements to initialize RAM. The loader does this initialization and therefore is not present at execution. You may initialize variables and arrays during execution via assignment statements or by READing into them.

2. FORMATs should not be read into during execution.

3. If you use the standard library I/O routines, do not OPEN DISK files on any LUNs other than 6, 7, 8, 9, 1Ø. (See Chapter 8, Input/Output, for an explanation of LUN.) If you need other LUNs for Disk I/O, reassemble $LUNTB with the appropriate addresses pointing to the Disk driver routine.

A library routine, $INIT, sets the stack pointer at the top of available memory (as indicated by the operating system) before execution begins.

The calling convention is:

```
LD   BC,<return address>
JP   $INIT
```

CHAPTER 4 / THE LINKER

The Linker creates a complete, executable program out of your relocatable object file. After it does this, you can save the program on diskette as a TRSDOS command file that you can load and run at the TRSDOS Ready prompt.

In creating an executable TRSDOS command file out of your program, the Linker automatically searches the system subroutine "subprogram" library (FORLIB) and loads the library routines needed to satisfy any undefined global references (that is, special calls that have been generated by the object program to the subroutines in the FORLIB library).

In the Introduction is a list of seven subroutines needed to run SAMPLE/REL -- the relocatable object file you created with your Compiler.

Subroutines Referenced:

$I1	$INIT	$L1
$T1	$W2	$ND
$EX		

These subroutine names are called global symbols. The globals are "undefined," because your relocatable object file does not know where in memory to find them. When you use the Linker to create the command file, it automatically inputs the required globals and their memory addresses and thereby links the globals to the program.

Running the Linker

To use the Linker, type:

L8Ø <ENTER>

This loads and executes your Linker. When it is ready to accept your commands, it prompts you with an asterisk. To exit the Linker, press <BREAK>.

LINKER COMMANDS

You can use several formats when linking a program:

command filename-N,object filenames-switch (es)

command filename-N -- Creates a TRSDOS command file. This is optional. If you do not specify an extension, the Linker uses CMD. When using command filename-N, the -E switch is required.

object filename -- This is the relocatable object file(s) that needs to be loaded. If omitted, the previously loaded object file(s) is used. If you do not specify an extension, the Linker uses REL.

switch -- Specifies what action the Linker takes with the object file. If omitted, the -U switch is used.

Remember, all filenames must be in standard TRSDOS filename format. (For further information, see Chapter 1.)

Examples

```
ELEVEN <ENTER>
```

inputs ELEVEN/REL. Since you do not specify a switch, the Compiler displays the file using the -U format. (This is discussed later in this chapter.)

FILE-N,FILE-E

inputs FILE/REL into the Linker and creates a command file (FILE/CMD).

PROG, SUBPROG

inputs PROG and SUBPROG into the Linker.

Switches

You may use several switches when you enter a Linker command to specify actions affecting the linking process. You must precede Link switches with a hyphen (-).

The Link switches are:

-R reset
-P specifies program area
-D specifies data area
-N saves command file
-U lists origin and end of program; undefined globals
-M lists origin and end; defined and undefined globals
-E exits LINKER and returns to TRSDOS
-S searches file for globals

-R

Reset. This switch puts the Linker back into its initial state. Use -R if you load a file by mistake and want to restart. You can also use this switch to delete the program

currently in the Linker and load another program. -R takes effect as soon as you enter it in a Linker command. For example, assume you input this file into the Linker:

INVEN1 <ENTER>

Now, if you want another file, type:

-R <ENTER>

and then input your other file:

INVEN2 <ENTER>

-E or E:name

Exits the Linker and returns to TRSDOS Ready. The system library is searched on your disk to satisfy any existing undefined global (subroutines in your library).

The optional form, E:name, returns you to the starting address of name rather than TRSDOS Ready. name must be a previously defined global symbol.

You almost always use this switch when using the N switch (see N for an example).

filename-N

Saves the program on diskette under the filename you selected (with a default extension of /CMD). You must also specify the -E (Exit) switch when you use the -N switch.

For example:

```
FAN-N,SAMPLE-E <ENTER>
```

inputs SAMPLE/REL into memory, creates a command file called FAN/CMD, and exits to TRSDOS Ready. Once you create it, you can execute FAN/CMD at the TRSDOS Ready prompt.

Be sure to use the N switch at the beginning of the Linker command. For example:

```
SUBNAME,PROG-N,PROG-E
```

does not work properly, whereas

```
PROG-N,SUBNAME,PROG-E
```

works.

-P and -D

Let the origin(s) be set for the program being loaded. (The origin is the absolute address of the beginning of the program area on your diskette.) As soon as you enter the switch(es), the computer reads -P and -D, but they have no effect on the program that is currently loaded. If you do not use the -P and -D switches, your program is loaded into memory beginning at the address of 3ØØØ (default). The data portion precedes the program portion. -P and -D let you load the program into different memory addresses.

```
SUBNAME,PROG-N,PROG-E
```

does not work properly, whereas

```
PROG-N,SUBNAME,PROG-E
```

works.

-P and -D

Let the origin(s) be set for the program being loaded. (The origin is the absolute address of the beginning of the program area on your diskette.) As soon as you enter the switch(es), the computer reads -P and -D, but they have no effect on the program that is currently loaded. If you do not use the -P and -D switches, your program is loaded into memory beginning at the address of 3ØØØ (default). The data portion precedes the program portion. -P and -D let you load the program into different memory addresses.

Use this form:

-P: address or -D: address

where address is the desired origin in the current base you are using. (The default base is hexadecimal. -O sets the base to octal, -H to hexadecimal.)

If you do not enter -D, your data areas are loaded before your program areas. If -D is given, all data areas are loaded, starting at the data (area) origin, and the program area at the program origin.

Examples

```
ELEVEN-P:4ØØØ
```

loads ELEVEN/REL, beginning at memory address 4ØØØ. Since you do not specify -D switch, the program portion precedes the data.

For example:

ELEVEN-U <ENTER>

displays the origin and end of the program and data areas as well as all undefined globals of ELEVEN/REL.

-M

Lists the origin and end of the program and data area, all defined globals and their values, and all undefined globals, followed by an asterisk. If you enter a -D, your screen displays program information. Otherwise, the program is stored in the data area.

-S

Whenever the Linker saves a FORTRAN program (-N), it automatically searches the FORTRAN library and links the appropriate routines. The -S switch forces a search of the system library also.

Section II

The FORTRAN Language

Part 2

THE FORTRAN LANGUAGE

INTRODUCTION

FORTRAN is a problem-oriented programming language designed for both experienced programmers and beginners. (FORTRAN is an acronym formed from the words FORmula TRANslation. This version is an extension to the ANSI Standard FORTRAN (X3.9-1966).) It is simple enough for new programmers to understand, but has many advanced features that make it a powerful calculating tool.

Your FORTRAN package is arranged so that your Model 4 can use each segment of the package efficiently, saving most of the memory for your actual program.

Just as you do in other high-level computer languages, you must follow precise syntactical procedures, such as the naming of variables, the order of statements, and the handling of input and output.

This manual explains these procedures and gives short examples describing their use. It is not a tutorial of FORTRAN programming, but a reference manual that shows you how F8Ø treats the FORTRAN language.

Chapter 5 describes the structure of FORTRAN programs, Chapters 6, 7, 8, and 9 describe the language features, and Chapters 1Ø and 11 provide an alphabetic list of FORTRAN statements and functions, along with their syntax and use.

CHAPTER 5 / FORTRAN PROGRAM STRUCTURE

The FORTRAN Character Set

All FORTRAN programs consist of one main program and any number of subprograms. The program units are made up of an ordered set of lines, or "statements." You write statements with the FORTRAN character set, which consists of:

letters:

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,$

numbers:

Ø,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

(the letters A-F are only for hexadecimal representation), and the special characters:

	blank
=	equal sign
-	minus sign
+	plus sign
*	asterisk (multiplication)
/	slash (division)
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point
**	double asterisk (exponentiation)

FORTRAN statements are on 8Ø-character lines that have the following format:

Column 1-5	statement labels
Column 6	continuation marker
Column 7-72	FORTRAN statements
Column 73-8Ø	identification field

Statement labels are integers in the range 1 to 99999. Blanks before or after the number are insignificant.

Labels must be unique in the program, that is, there cannot be more than one label with the same value in the same program or subprogram. Labels are only valid on the initial line of a statement, not on continuation lines (see the following).

The actual FORTRAN code can begin with Column 7 and may extend to Column 72. If it is necessary to go beyond that, you can continue the line on the next line by placing a character in Column 6 and finishing the statement.

Often it is convenient to mark the continuation lines by placing a number in Column 6, but any character will do. You may have as many continuation lines as necessary to complete the line. The computer ignores statement labels on continuation lines.

The computer also ignores the identification field, Columns 73-80; thus, you can use it for any purpose you choose. For example, you can number your program lines and put the line numbers at that point, or you can insert short comments in this field. Generally, however, this field is left blank.

Figures 1 shows a sample FORTRAN program written on a "coding sheet" -- a tabular form with columns, lines, and fields. Following Figure 1 is a list that gives the meaning of each statement of the program.

FORTRAN Statements

FORTRAN statements are made up of constants, for example, 6.2; variables, for example, AVE; and FORTRAN command words, for example, READ. The two broad categories of FORTRAN statements are executable and nonexecutable.

EXECUTABLE STATEMENTS

Executable statements describe a certain action (or chain of actions) that the computer is to perform. They can be "replacement" statements, "control" statements, or I/O statements.

Replacement Statements

Replacement statements set a variable (variables are a symbolic way of representing a number) equal to some value.

This value can be a number, another variable, or a list of variables and operators. For example:

```
A = 34.5
```

sets A equal to 34.5.

Control Statements

The computer normally executes statements in a program in the order in which they appear. Control statements change that natural flow (or sequence of events) so that certain statements are executed before other statements and other statements are repeated. Control statements also mark the end of the program. For example:

```
GO TO 3Ø
```

transfers control of the program to the statement labeled 3Ø. Control statements include:

```
CALL
DO
GO TO
IF
RETURN
```

These statements are related to control statements:

ASSIGN
CONTINUE
END
INCLUDE*
PAUSE
STOP

* INCLUDE is actually a "pseudo op" (that is, it is not executable).

Input/Output Statements

The computer uses Input/Output (I/O) statement to communicate with you and with data stored in disk files. One example is the WRITE statement:

```
WRITE(5,1Ø) A
```

This statement instructs the computer to output the value of the variable A to your screen. (5 is the device to which the output is to be sent; it is the device number for the screen. 1Ø is a FORMAT statement label that is discussed in Chapter 8.) I/O commands include:

OPEN
REWIND
ENDFILE
INP
OUT
READ
WRITE

NONEXECUTABLE STATEMENTS

Nonexecutable statements describe to the computer the nature and storage of data and variables. Included in the nonexecutable category are "type specification" statements, "array declarators," "DATA" statements, "FORMAT" statements, storage definers, and "definition" statements.

Type Specification Statements

Some "types" of variables represent integer numbers, others represent real numbers, and others represent numbers that, for reasons of precision, must take up extra storage space. The next chapter discusses the different types of variables.

Variables beginning with certain letters are by default a certain variable type. For instance, all variable names beginning with the letters I-N represent integers. But sometimes, for reasons of clarity, it is necessary to have an integer variable that, for example, begins with A. You do this with a "type declaration" statement. For example:

```
INTEGER A
```

defines A as an integer variable. Type declarators include:

```
BYTE
DOUBLE PRECISION
IMPLICIT
INTEGER
INTEGER*4
LOGICAL
REAL
```

Array Declarators

You can express several values as a group of numbers, called an array. FORTRAN requires, for storage allocation purposes, that you "declare" the size of the array before the program begins. For example:

```
DIMENSION A(1Ø)
```

declares that the array A consists of 1Ø elements, or members. You can also declare array dimensions in conjunction with type declarators. For example:

```
INTEGER A(1Ø,1Ø)
```

declares that array A consists of 1ØØ (1Ø X 1Ø) members. Each member is an integer.

DATA Initialization Statements

You can set the value of variables with I/O statements, for example, READ, and with replacement statements; both are executable. DATA statements, which are nonexecutable statements, can also set the value. For example:

```
DATA A/34/
```

FORMAT Statements

You can use FORMAT statements to specify the appearance of input or output data. For example:

```
1Ø   FORMAT(1ØX,I2)
```

Data processed through this statement takes on the shape specified by 1ØX and I2, which are called FORMAT specificators and are discussed in later chapters.

Storage Definers

You may need to define two or more variables to be in the same memory location. You can do this with storage defining statements. For example:

```
EQUIVALENCE (A,B)
```

puts A and B in the same storage locations. The two storage-defining statements are:

```
COMMON
EQUIVALENCE
```

Definition Statements

A FORTRAN program can consist of a "main" program and one or more "subprograms," and you must "title" these program units. For example:

```
SUBROUTINE CREDIT(A,B,C)
```

declares that your statements which follow are part of the SUBROUTINE CREDIT that uses the variables A, B, and C. Definition statements include:

```
BLOCK DATA
FUNCTION
SUBROUTINE
PROGRAM
```

The FORTRAN Program

The "source" program brings together your individual FORTRAN statements. It is a listing of the FORTRAN statements. The computer executes the statements in the order that they are listed, except when it executes a control statement that causes "branching" of the program.

PROGRAM BRANCHING

These statements generate branching within the program (intraprogram branching), cause the program to branch to an external program (subprogram branching), or terminate the program.

Intraprogram Branching

GO TOs and DO loops cause branching inside the main program. This eliminates the need for typing the same set of statements again and again. For example:

```
20    A = A + 1.0
      IF(A.LT.B) GO TO 20
```

The variable A is incremented by one and then compared to B. Control transfers to statement 2Ø until A is no longer less than (.LT.) B.

DO loops are specialized forms of GO TOs. The DO loop provides a simple way of looping a program through the same commands for a given number of times.

Consider this example:

```
        DO 1Ø I=1,1Ø,2
        N=N+I
  1Ø    CONTINUE
```

Here the loop is executed 5 times (I is equal to 1,3,5,7, and 9). Statement 1Ø is a CONTINUE statement. This is a common way of marking the end of a loop, because the CONTINUE statement causes no action to be taken by the computer. (For more information on DO, see Chapter 1Ø.)

Subprogram Branching

Branches to subprograms are somewhat different. When control transfers to a function or subroutine, the computer executes statements in the subprogram. After executing all the statements, control returns to the main program.

Execution of the main program begins at the point immediately following the call to the subprogram. Branching to subprograms is a part of "segmenting" programs, which is discussed in Chapter 9.

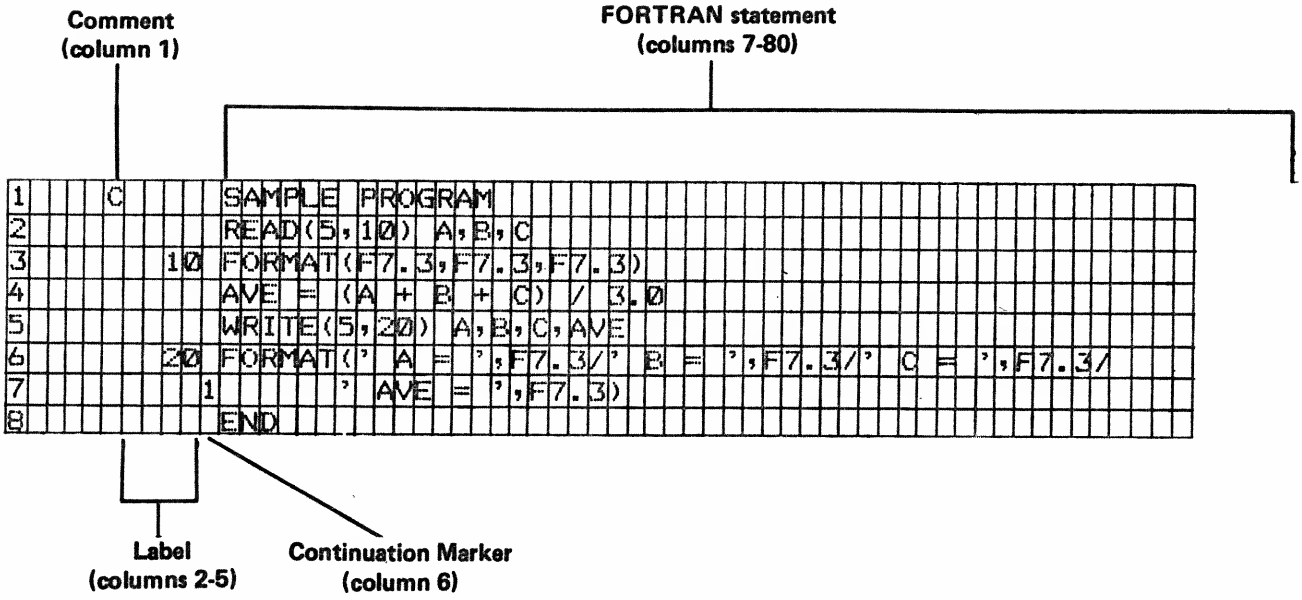
Order of Statements

To execute properly, the statements in your program or subprogram must be in the following order:

1. Definition statements

2. Type specification statements, array declarator statements, and the EXTERNAL statement (discussed in Chapter 1Ø)
3. Storage definer statements (COMMON must precede EQUIVALENCE)
4. Data initialization statements
5. All executable statements
6. The END statement

The FORMAT and INCLUDE statements can appear anywhere in the program.


Comment
(column 1)
FORTRAN statement
(columns 7-80)
C SAMPLE PROGRAM
READ(5,10) A,B,C
10 FORMAT(F7.3,F7.3,F7.3)
AVE = (A + B + C) / 3.0
WRITE(5,20) A,B,C,AVE
20 FORMAT(' A = ',F7.3/' B = ',F7.3/' C = ',F7.3/
1 ' AVE = ',F7.3)
END
Label
(columns 2-5)
Continuation Marker
(column 6)

Line Number	Interpretation
1	This is a comment line. Note the "C" in Column 1.
2	This is an I/O statement. It tells the computer to READ in three values from the keyboard and store them into the variables A, B, and C. The (5,1Ø) means that the input is to come from I/O unit 5, the keyboard, and the data appears as it is described by the format in Statement 1Ø.
3	This is a FORMAT statement. It describes how data should look when it is read in or printed out. In this case, it applies to the READ statement label in Columns 4-5. The F7.3 is called a field specification. It implies that the data processed through it is Floating point (that is, it contains a decimal point), it is seven characters long, and it has three characters to the right of the decimal point.
4	This is a replacement line. It tells the computer to set the variable AVE equal to the sum of the variables A, B, and C, divided by the constant 3.Ø. Algebraically, this is the same as $\frac{A + B + C}{3.Ø}$
5	This is another I/O statement. It tells the computer to WRITE the values stored in the variable A, B, C, and AVE to the I/O unit 5 (the screen), using

the format specification in Statement 2Ø.

6 This is another FORMAT statement. It is labeled 2Ø and, in this case, is used for the output from the above WRITE statement. The F7.3 has the same meaning as in Line 3 (Statement 1Ø), and, in addition to that descriptor, this specification has strings enclosed in quotes. These are called "literals" and print out along with the variable values. The slashes (/) tell the computer to skip a line before printing the next record (set of data).

7 This is a continuation line. It contains the remainder of the line directly above it. Note the 1 in Column 6.

8 This is a control statement. It signifies the final step of the program (the "logical" and "physical" end of the program).

When you execute this program, the READ line stops and waits for you to input values for A, B, and C. The format line allows each value to be a maximum of seven characters and to have a maximum of three decimal digits. Type:

```
12.5, 9.55, 1Ø.Ø
```

This data is interpreted as 12.5, 9.55, and 1Ø.Ø. The output for this program using this data is:

```
A = 12.5ØØ
B =  9.55Ø
C = 1Ø.ØØØ
AVE = 1Ø.683
```

CHAPTER 6 / DATA

You can represent data as either constants or variables. A constant is the data itself. Examples of constants are:

34.4 2 'BILL' -32.78356

A variable is a symbolic name that represents the number. Examples of variables are:

AVE M123 ABCDEF Z

A variable is composed of one to six alphanumeric characters, the first of which must be a letter.

Types of Data

Each type of data has its own form and precision. The eight types are integer, extended integer, byte, real number, double precision number, logical, literal, and hexadecimal number. (For further information, see "Storage Format," Appendix G.)

INTEGER

An integer is a whole number (no decimal point), the value of which may be in the range -32768 to 32767. The following are integers:

-32765 -85 1 2Ø148

Integers are sometimes advantageous to use since they occupy the least amount of storage space (only two bytes). Also, the operations using integers are the fastest.

By default, all variable names that begin with the letters I-N are integer variables. For example:

```
NUM
```

is an integer variable, because it begins with N. You can changes this by the declaration statement INTEGER:

```
INTEGER ACCTNO, XMAX
```

This line declares the variables ACCTNO and XMAX to be integers.

EXTENDED INTEGER

An extended integer is a whole number the value of which is in the range -2,147,483,648 to 2,147,483,647. The following are extended integers:

```
76543215        -67839624
```

An extended integer occupies four bytes of storage space. There are no default extended integers, so you must "declare" them with the INTEGER*4 statement:

```
INTEGER*4 AVE,NUM
```

This line declares AVE and NUM to be extended integers.

BYTE

A byte is a number the value of which is in the range -127 to 128. THe following are bytes:

```
-115            5               94
```

A byte consumes one byte of storage space. Bytes are often used as arrays for string data (each string character

requires one byte). There are no default bytes, so you must "declare" them with a BYTE statement. For example:

```
BYTE AVE,NUM
```

declares AVE and NUM to be bytes.

REAL NUMBER

A real number contains a floating decimal point, and its value lies between 1Ø**-38 and 1Ø**38 (** means to the power of).

In real numbers, the decimal point "floats"; in integers, the decimal point is fixed at the extreme right of the number. Examples:

```
137.56    Ø.Ø14    5.Ø    7.234
```

These are approximations that are precise to seven significant digits, and they each occupy four bytes of storage space. They can be represented in decimal form

```
345.322
```

or in exponential notation

```
2.4842E-1
```

The above example is equal to 2.4842 x 1Ø to the power of -1, which is equal to Ø.24842. Variables beginning with the letters A-H and O-Z are by default real variables. For example:

```
AVE
```

is a real variable. You can define more variables as real by using the REAL declarator. For example:

```
REAL INC,NUM
```

declares INC and NUM to be real numbers.

DOUBLE PRECISION NUMBER

A double precision number may be in the same range as a real number, but it is precise to 16 significant digits. It occupies eight bytes of storage space and must be represented exponentially. For example:

```
27.4332Ø92134151DØ (note, D instead of E)
```

The symbol D denotes double precision exponential. Therefore, 27.4332Ø9134151DØ represents 27.4332Ø9134151 X 1Ø to the power of Ø, which equals 27.4332Ø9134151 X 1, or 27.4332Ø9134151.

```
23.4949433D23
```

represents the value 23.4949433 X 1Ø to the power of 23.

IMPORTANT NOTE: If you do not use the D notation when typing a double precision number, FORTRAN assumes it is a real number and rounds it off to seven signficant digits.

Other Examples

```
123456789Ø1234DØ          Ø4Ø1249671229D-12
```

There are no double precision variables by default, so you must define them in declaration statements. For example:

```
DOUBLE PRECISION AVE,NUM
```

declares AVE and NUM to be double precision variables.

LOGICAL

A logical is a one-byte representation of the values of "true" and "false" that you can use in logical arguments. (For further information, see Chapter 7.)

As constants you can represent them as:

.TRUE. and .FALSE.

These are stored (respectively) as:

-1 and Ø

You can also use them as integers with the range of -128 to 127. You must define logical variables by means of declaration statements. For example:

LOGICAL L1,ANSWER

declares the variables L1 and ANSWER to be logical variables.

LITERAL

A literal is a string composed of any characters in the FORTRAN character set. As constants you can represent them by enclosing them in single quotes. For example:

'BILL' 'HELLO'

You can also precede the string with the string's length followed by the letter H. For example:

4HBILL 5HHELLO

When stored in a variable, each character occupies one byte of space; so the size of the variable determines the size of the string.

For example, integer variables occupy two bytes of storage space. Therefore, the largest integer string that any integer variable can hold is two characters long. For example,

```
N = 'ME'
N = 'MEN'
```

both set N equal to ME.

When using replacement statements, you can only assign literals to bytes, logicals, or integer variables. You cannot use a replacement statement to assign them to extended integers, real numbers, or double precision variables.

When you use replacement statements, the largest string that one variable can hold is two characters. Often it is necessary to use strings the lengths of which are longer than this.

You can eliminate this problem by storing strings in variable arrays. Arrays and their uses in strings are discussed later in this chapter.

HEXADECIMAL NUMBERS

You can use hexadecimal numbers (base 16) as constants by preceding them with the letter X or Z and enclosing the number (one to four digits long) in single quotes.

Generally, you can use integers to store hexadecimal numbers. However, you can store hexadecimal numbers with real variables only through DATA statements. For example:

```
N = X'AØ'
DATA A/X'AØ64'/
```

store the hex value AØ (decimal 16Ø) into the integer variable N and the hex value AØ64 (decimal 41Ø6Ø) into A.

IMPLICIT Declaration

It is often convenient to define many variables in one statement that begins with the same letter or range of letters. Since it may become cumbersome to define each variable in a type declaration statement, FORTRAN contains the IMPLICIT statement.

The IMPLICIT statement defines all variables that begin with a certain letter or range of letters to be of a certain variable type. For example:

```
IMPLICIT INTEGER(A-H,O-Z), REAL(I-N)
```

declares all variable names beginning with the letters A-H and O-Z to be integers, and all variables beginning with the letters I-N to be real variables. This is the exact opposite of the default condition. In another example:

```
IMPLICIT DOUBLE PRECISION (D)
REAL DEV
```

declares all variables that begin with D, with the exception of DEV, to be double precision.

Scalar and Array Variables

The two types of variables are scalar and array. A scalar may contain only one value; an array may contain many values.

A scalar is identified by a single symbolic name. For example:

```
TOT
```

An array is identified by a symbolic name followed by a parenthetical subscript. For example:

```
ITEM(3)
```

Both kinds of variables receive their values when the program is running or through DATA statements (see "Assigning Values to Variables").

You do not need declaration statements with scalars unless you want to declare them to be a different type from their default type (for example, you might want to declare A as INTEGER A). When you run the program, FORTRAN automatically reserves storage space for them.

If you plan to use arrays, on the other hand, you must use a special "dimension" declarator statement to reserve storage space for them. You can append the variable list following a type declarator (for example, INTEGER or REAL) to define its size. For example:

```
REAL NUM(3,3,3)
```

defines NUM as a real three-dimensional array with 27 (3 X 3 X 3) elements. You can also define an array size with a DIMENSION statement. For example:

```
DIMENSION ARRY(2Ø)
```

defines ARRY as a one-dimensioned array with 2Ø elements.

ACCESSING THE ARRAY ELEMENTS

Arrays can have "length" and/or "height" and/or "depth," depending on the number of dimensions. The computer stores an array in memory in a single-dimensioned list of the elements of the array. (For further information, see Appendix G.)

You can access an array by its subscript. Subscripts are integer constants or variables, or algebraic expressions the values of which are integer. A subscript can describe an individual element.

For example, you can think of a one-dimensional array as a list. You can address the fifth element in the list as ARRY(5). A two-dimensional array is then a "table," and you can address an element in Row 4 and Column 3 as ARRY(4,3).

Using the same analogy, you can think of a three-dimensional array as a "book" of "tables." ARRY(2,4,3) describes the element in Row 2, Column 4, Page 3.

The following rules apply to subscripts:

- The number of subscripts in an array element description must be the same as the number specified in the dimension declarator. For example:

```
DIMENSION A(1ØØ,5)
A(1Ø) = 5.Ø
```

  is not valid.

- If you use an algebraic expression as a subscript, it must have an integer value when evaluated.

- Subscripts themselves cannot have subscripts.

Assigning Values to Variables

You can assign values to variables with replacement statements and with DATA statements. The following replacement statement

```
A = 23.54
```

sets the value of A equal to 23.54. In this statement, 23.54 is an "expression." The following replacement statement

```
A = A + B
```

sets A equal to the value of A + B, or more exactly, A is "replaced" by the sum of A and B (B remains unchanged). (For further information on expressions, see the next chapter.)

DATA statements are nonexecutable assignment statements with the following syntax:

```
DATA variable/data/variable/data/...
```

For example:

```
DATA A/12.4/B/13.2/C/200.3/J/0/
```

sets A = 12.4, B = 13.2, C = 200.3, and J = 0.

You can assign values to arrays in this way also. For example, consider a five-element array, NUM. You can assign values through this statement:

```
DIMENSION NUM(5)
DATA NUM/12,42,23,23,10/
```

If you want to initialize all the elements of an array to the same value, place an asterisk (*) after the number of elements in the array, following by the initial value.

For example:

```
DATA ARRAY/100*0.0/
```

sets all 100 elements of ARRAY equal to zero.

It is sometimes convenient to use DATA statements to give string data values to arrays. For example:

```
BYTE WORD(5)
DATA WORD/'T','U','L','I','P'/
```

stores the string TULIP into the byte array WORD, so that:

```
WORD(1) = T
```

```
WORD(2) = U
WORD(3) = L
WORD(4) = I
WORD(5) = P
```

COMMON Variable Storage

COMMON DATA STORAGE

Sometimes it is necessary to store two different variables in the same memory location. You can use the COMMON statement to pass data to and from subroutines. (For further information, see "Segmenting Programs," Chapter 9.)

EQUIVALENCE Statement

COMMON statements put variables in the same memory location for use by subroutines. Sometimes it is convenient to put variables into the same memory location for use in the same program unit. You might do this to save memory space or to give two variables the same value. You can do so with an EQUIVALENCE statement. For example:

```
EQUIVALENCE (A,B,C)
```

puts A, B, and C into the same storage unit.

CHAPTER 7 / EXPRESSIONS

Expression is another word for data. It can be simple:

```
24          X          ABS(5)
```

or complex:

```
5 + 6          X + Y          ABS(5) - 24
```

A simple expression consists of a single operand. This may be a constant, a variable, or a function. (For further information on functions, see Chapters 9 and 11.)

A complex expression consists of two or more operands separated by operators. Operators are characters that tell the computer what to do to the operand.

The three kinds of complex expressions are arithmetic, relational, and logical.

Arithmetic Expressions

An arithmetic expression consists of constants, variables, and functions connected by arithmetic operators. The arithmetic operators are:

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

In addition, the + and - operators serve as positive and negative indicators.

Important Note: You cannot place two operators side by side. To express an operation that uses the negative value of a number, you must enclose the minus sign and the number in parentheses. For example:

```
A*(-B)
```

EXAMPLES OF ARITHMETIC EXPRESSIONS

```
H/I+B
5/4+3.2
3*5
PI*R**2
TOTAL+((SQRT(A-4.5)/C**D)-AVE(I,J))
```

When an arithmetic expression is longer than one or two operands, it is difficult to determine the order of execution. For example, is 1Ø+2Ø/5 equal to 14 or 6?

For this reason, levels of evaluation hierarchy have been established. The levels from highest to lowest are:

Functions
Parentheses
Exponentiation
Multiplication and division
Addition and subtraction

Within each level, the computer evaluates the expressions from left to right, except for parentheses, which it evaluates from the innermost set to the outermost.

Important Note: Whenever you use integers for division, the computer truncates (chops off) the decimal portion of the quotient, leaving only the whole number. For example, the computer evaluates

```
3Ø/4
```

as 7, but:

```
3Ø.Ø/4.Ø
```

as 7.5.

Example 1

```
(6+9)*(5-4)/(2+1)
```

Here the computer first calculates the values of 6+9, then 5-4, and, finally, 2+1. Then it multiplies 15 times 1 and divides by 3. This statement is identical to the algebraic expression:

$$\frac{(6+9)(5-4)}{(2+1)}$$

Here you can see the expression equals 5.

Example 2

```
6.Ø**2+7.Ø*3.Ø-3.Ø/(5.Ø*(1.Ø+2.Ø))
```

This equation, algebraically, looks like this:

$$6.Ø^{2} + 7.Ø \times 3.Ø - \frac{3.Ø}{5.Ø \times (1.Ø + 2.Ø)}$$

Upon evaluation, the expression equals 56.8.

Example 3

```
12.Ø*4.Ø-SQRT(25)/(6.5-5.5)
```

is evaluated as 43.Ø.

When you use different variable or number types (modes) in the same expression, the resulting number is in the mode of the data item with the highest hierarchy. This hierarchy is as follows, from highest to lowest.

Double precision
Real
Extended integer
Integer
Logical
Byte

The validity of statements containing mixed modes depends on the operations performed. For example, adding a double precision number to an integer results in a double precision number. However, the sum is not accurate past the decimal point. For instance, the computer evaluates:

```
2.99293944DØ + 35/12
```

as 4.99293944, which represents an error of 15% (the correct answer is approximately 5.9Ø96Ø61Ø7).

Relational Expressions

Relational expressions contain two arithmetic expressions connected by a "relational" operator. The six relational operators are:

.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not Equal to
.GT.	Greater than
.GE.	Greater than or equal to

The computer evaluates a relational expression as "true" or "false." It sets true statements equal to -1; false statements equal to Ø. The computer first evlauates the

arithmetic expressions, each in its own mode, on either side of the relational operator. Then it compares the two, using the highest mode.

Relational expressions can assign values to logical variables. For example:

```
L = 1.GT.5
```

sets L equal to Ø (false) since 1 is not greater than 5.

More often, however, you use relational expressions in "logical IF" statements. Logical IFs contain the word IF followed by a relational expression enclosed in parentheses, followed by an executable statement.

For example:

```
IF(1.LE.(5.Ø/3.Ø)) GO TO 2ØØ
GO TO 1ØØ
```

In such statements, the computer first determines if the expression is true or false. If it is true, the computer executes the next statement. Otherwise, the computer executes the next line.

In the above example, since 1 is less than 5.Ø/3.Ø, control transfers to Statement 2ØØ. Otherwise, the program continues to the next line, in this case another GO TO.

Additional logical IF examples:

```
IF(A-B.GT.(D**2)) C=A-B

IF(D.NE.E) WRITE(5,5) D,E

IF(A.LT.Ø.Ø) CALL SUBl(A,B,C)
```

Logical Expressions

Logical expressions are those the computer can evaluate as either true or false. They can be simple or complex. A simple logical expression consists of a single logical operand. This may be a logical variable or a relational expression. For example:

```
.TRUE.        L1        5.GT.A
```

A complex logical expression consists of two or more logical operands separated by one or more logical operators.

The four logical operators are AND, OR, NOT, and XOR. The computer evaluates them in the following way (A and B represent logical operands):

A.AND.B	true only if A and B are true
A.OR.B	true if either A or B is true
.NOT.A	true only if A is false (note that this operator requires only one operand)
A.XOR.B	true only if one of the operators is true and the other false

(**Note:** The operations of logical operators are based on bit-by-bit comparisons and Boolean logic. If you are interested in how they work, consult a good mathematics or computer logic book.)

These are examples of logical operations.

```
X.AND.Y        5.GT.X.OR.3.EQ.Y
```

Logical expressions assign values to logical variables:

```
LG1 = A.LT.B.OR.A.GT.C
```

but often they represent multiple relational expressions in logical IFs:

```
IF(A.EQ.B.AND.B.EQ.C) A=D
```

The above statement says that only if A is equal to B and B is equal to C should A be set equal to D.

```
IF(M.LT.N.XOR.O.EQ.P.) A=B+M**2
```

The statement contained in the parentheses is only true if M is less than N and if O is not equal to P, or if M is greater than or equal to N and O equal to P (that is, it is true only if one of the expressions is true).

The following hierarchy exists for the valuation of logical expressions:

```
Parentheses
Function references
Multiplication and division
Addition and subtraction
Relational operations
.NOT.
.AND.
.OR., .XOR.
```

where the computer executes the operations within the same level from left to right. If you compare this list to the arithmetic hierarchy, you can see that the computer first evaluates parentheses, then arithmetic expressions, relational expressions, and finally logical operations.

Example

```
IF(AVE.EQ.SUBAVE.OR.(A/D-1).LE.A.AND.L1)
```

In this statement, the computer evaluates A/D-1, then expression AVE.EQ.SUBAVE. After that, it compares the value

found by A/D-1 to A. Next it compares that value to L1. Finally it compares that value to the value found by AVE.EQ.SUBAVE.

Note: It is invalid to have two logical operators adjacent to each other, unless the second operator is a .NOT.. For example:

```
L1.AND..NOT.L2
```

is a valid expression; if L1 is true and L2 is false, then the expression is true.

```
L1.AND..OR.L2
```

This is an invalid expression since it uses .AND..OR..

Also, a literal string may be only two bytes long in a logical IF.

CHAPTER 8 / INPUT/OUTPUT

Your F8Ø Compiler uses several different buffers, or "logical units," to pass data to and from external devices such as your disk drives, line printers, and terminal.

A buffer is a temporary storage area for information being transmitted from one unit to another. It compensates for the different speeds at which the units can handle data.

Unless otherwise specified, logical unit numbers (LUNs) are assigned as follows:

LUN 1 and 3-5 to the screen or keyboard
LUN 2 to the line printer
LUN 6-1Ø to the disk drives.

(Appendix B describes how to change the LUN assignments and add more LUNs.)

Data is input and output in groupings called records. Each record may consist of one or more subgroupings called fields. The length of each record depends on the program input/output commands and can be from 1 to 256 bytes long. (256 bytes is the size of the buffer.)

Opening a LUN

In order to perform input and output (I/O), you must open a LUN between the computer and the I/O device. You can let the computer do it automatically, or you can use the OPEN subroutine provided in the FORLIB FORTRAN library.

LETTING THE COMPUTER OPEN THE LUN

Your computer opens a LUN the first time it encounters a READ or WRITE to it. It sets the record length by default

to 128 bytes. When a disk file is involved, your computer assigns a default name to the file, based on the LUN used:

LUN	Default filename
6	FORTØ6/DAT
7	FORTØ7/DAT
8	FORTØ8/DAT
9	FORTØ9/DAT
1Ø	FORT1Ø/DAT

Consider the following program:

```
      READ(5,1Ø) A
1Ø    FORMAT(F8.3)
      WRITE(6,1Ø) A
      END
```

In this example, your computer opens LUN 5 (the keyboard) and stops execution of the program until you enter a value for A, using FORMAT 1Ø. (FORMATs are explained later in this chapter.) It then opens LUN 6 (a disk drive), gives it the default filename FORTØ6/DAT, and then writes A to it using FORMAT 1Ø.

USING THE OPEN SUBROUTINE

Letting the computer opens LUNs can be convenient, especially for output to your printer or for input and output with your screen. However, if you plan to input and output files to and from the disk, assigning default names limits the number of files you can use. This method also wastes disk space because the computer automatically sets the record length to 128 bytes and that amount is seldom needed.

The alternative is to use the OPEN subroutine. The syntax for addressing the OPEN subroutine is:

CALL OPEN(logical unit number,'filename', record length)

The OPEN subroutine opens a buffer between the computer and the device specified by the LUN. This can be any device but is generally a disk drive. After you do this, you have a file with the valid TRSDOS filename you specified to which you can write data. To access an existing file, you again use the CALL OPEN, and then you can read from it as well as write to it.

An example of a valid OPEN is:

```
CALL OPEN(6,'ACCT/FIL:1',40)
```

This opens a file called ACCT/FIL on Drive 1 accessed by Logical Unit 6. The file's records are 40 bytes long.

The record length must be an integer constant or variable the value of which is in the range 1 to 256. The record length must be large enough to store all your data. (The necessary length depends on your FORMAT statement and whether you are using direct or sequential accessing. The overhead involved in each is discussed later in this chapter.)

A file, once opened, remains open until an ENDFILE command closes it. For example:

```
ENDFILE 6
```

closes the file associated with LUN 6. If this command is not included, the computer closes the file automatically when it completes program execution.

INPUT AND OUTPUT FIELDS

You can "format" or "unformat" FORTRAN input/output operations. Formatted READs and WRITEs use a FORMAT statement, and unformatted READs and WRITEs do not use a FORMAT statement.

Unformatted Data

Unformatted READs and WRITEs read and send information in a "bit stream." That is, data is stored in memory exactly as read off the disk, and data is written to your diskette exactly as it is stored in memory.

To use unformatted READs and WRITEs, simply state READ (or WRITE), followed by the LUN in parentheses, followed by the variable list. For example:

```
READ(6) A, B, C
WRITE(7) A, B, C
```

input A, B, and C from LUN 6 and output them to LUN 7, unformatted.

Unformatted data is stored with a delimiter (separator) between every field but has no delimiter between records. Thus, it usually occupies less space on the diskette. However, you can use such data only for diskette input/output, and then it must always be reread as unformatted data.

Formatted Data

Formatted READ/WRITE's read and send data according to the format set in the FORMAT statement. A formatted READ/WRITE statement uses a label that indicates which FORMAT statement the computer should use.

FORMAT statements are nonexecutable and describe to the computer how a given data item should look on input or output. The syntax of a FORMAT statement is:

label FORMAT(specification list)

where label is a statement label and the specification list consists of "field" descriptor types, parentheses, carriage controls, and delimiters.

The 1Ø field descriptor types are A,D,E,F,G,H,I,L,P, and X fields and literal descriptors. Each has its own purpose and syntax, which is discussed on the following pages. Each descriptor in the specification, with the exception of X and H fields and literal descriptors, must have a corresponding variable listed in the READ or WRITE statement and be separated by a comma or slash (/).

Field Descriptors

The general syntax for the following descriptors, unless otherwise specified, is:

rFw.d

where r is an integer representing the number of times to repeat the current descriptor before using the next descriptor. The next letter, F in this case, is the field descriptor. w is the width of the field, and d is the width of the decimal portion of the field. (Note: Not all descriptors use all these dimensions.)

For example, the F descriptor is for floating point data; so an F14.5 field is broken down as:

```
w w w w w w w . d d d d d
- - - - - - - - - - - - -
```

where the w's represent the whole number portion of the number and the d's represent the decimal portion of the number.

Strings Stored in Variables

A field: rAw

The A desciptor inputs or outputs charactor strings (literal or Hollerith). The maximum width w for transmitting a variable depends on the storage requirements for that type of variable. Real variables require four bytes for storage; so the maximum usable width for the FORMAT description of a real variable is A4. Integers require two bytes of storage; so the maximum usable width is A2.

Input: If the field width w exceeds the storage size of the variable type into which it is to be read, the rightmost w characters are input into the variable (that is, the left characters are truncated). If the field width is smaller than the variable size, the data is left-justified inside the variable.

Examples

Field	Input data	Variable type	Stored as
A1	A	byte	A
A3	ABC	integer	BC
A1	ABCD	real	Abbb
A4	ABCD	real	ABCD
A7	ABCDEFG	real	DEFG
A8	ABCD EFG	dbl. pre.	ABCDbEFG

Output: If the field width w exceeds the storage size of the variable, the data is right-justified in the field. If the field width is smaller than the variable size, the farthest left w characters of the variable are written.

Examples

Field	Stored data	Variable type	Output as
A1	Ab	integer	A
A2	AB	integer	AB
A3	AB	integer	bAB
A3	ABCD	real	ABC
A6	ABCD	real	bbABCD
A7	bABCDEFG	dbl. pre.	bABCDEF

Double Precision

D field: rDw.d

The D descriptor inputs or outputs double precision numbers in the exponential form:

signØ.decimalDsignexponent

The total width w should be seven more than the decimal portion d to ensure that the signs and significant figures are kept.

Input: Data input under D specifications can be in one of three external formats. First, it can be in the actual exponential form. In this case, the computer converts the number to its real value, if possible, and the D portion of the descriptor is ignored. Second, data can contain a decimal point, in which case it is read in as is, and again the D ignored. Third, it can be an integer with the d portion determining the decimal location (the blanks are interpreted as zeroes). For integer input, the number must be right-justified within its field.

Examples

Field	Input value	Stored value
D19.12	-Ø.4999291448329D3	-499.9291448329
D15.8	bbb19.9392Ø491Ø	19.9392Ø491Ø
D15.8	bbbbl9283928571	192.83928571

Output: Data output under D specifications is right-justified in its field. The computer converts the internal value to D notation and rounds it to fit the field.

Examples

Field	Internal value	Output
D14.7	4932.8588321	bØ.4932859D+Ø4
D18.11	-.85893917477	-.85893917477D+ØØ
D15.8	.ØØ3Ø83824888	bØ.3Ø838249D-Ø2

E field: rEw.d

The E field descriptor inputs or outputs numbers in the exponential form:

signØ.decimalEsignexponent

The width w should be seven more than the decimal portion to ensure that all signs and significant figures are kept.

Input: Data input under E specificatiions can be in one of three forms. First, it can be in integer form with the d portion determining the decimal size. Second, it can be real and input as is (that is, the decimal point in the field description is ignored). Third, it can be exponential with the number converted to real and the d ignored. For integer input, the number must be right-justified within its field.

Examples

Field	Input value	Stored value
E12.5	-Ø.48293EbØ3	-482.933
E13.6	bbbb9348299bb	934.8299
ElØ.3	bbbbl24.43	124.43

Output: Data output under E specifications are right-justified within the field. The computer converts the internal value and rounds the number to fit the field.

Examples

Field	Internal value	Output
E12.5	823.Ø993	bØ.8231ØEbØ3
E14.7	-.ØØØ38414	-Ø.3841ØØE-Ø3
E14.5	923.9	bbbØ.9239ØEØØ3

Real and Double Precision

F field: rFw.d

The F FORMAT field inputs and outputs real and double precision numbers. The general form of values processed by this field is:

signinteger.decimal

Input: Data input under F specifications can be in one of three forms. First, it can be in integer form where the d portion determines the decimal size. Second, it can be real and input as is (that is, the decimal point in the field description is ignored). Third, it can be exponential. In this case, the computer converts the number to real and again the d is ignored. For integer input, the number must be right-justified within its field.

Examples

Field	Input value	Stored value
F7.2	b45.99b	45.99Ø
F6.1	b94754	9475.4ØØ
F12.3	bb-Ø.8583lE4	-8583.1

Output: Values output in an F field are right-justified in their field. The field must be large enough to allow the integer portion of the number to be output in its entirety. If the number is too large, a portion of the data is output with an asterisk in the middle. The decimal portion, if too large for the field, is rounded to fit the field. After giving you an FW Compiler runtime error, the Compiler outputs the number right-justified in the field.

Examples

Field	Stored value	Output value
F6.2	392.8	b392.8Ø
F8.3	3943.9Ø1	3943.9Ø1
F8.5	Ø.9492E-Ø2	bb.ØØ949
F1Ø.7	131.93492	1319349Ø*6

Real and Double Precision

G fields: rGw.d

The G descriptor inputs and outputs real and double precision data. This descriptor acts as an F descriptor for some data and as an E descriptor for other data, depending on the size of the data.

Input: Data input under G specifications can be in one of three forms. The first is integer form in which d determines the location of the decimal point. Second, it can be real. The d portion is ignored, and the number is

input as is. Third, it can be exponential. The computer converts the number, if possible, to its real form, and the d is ignored. For integer input, the number must be right-justified in its field.

Examples

Field	Input value	Stored value
G1Ø.3	bbb29849bb	2984.9ØØ
G12.5	bbb193.94bbb	193.94
G13.6	Ø.99434EØ5bb	99434.Ø

Output: The format of G-specified output depends on the magnitude of the data to be output. If the number has d or fewer significant digits, it is output as Fw.d. However, if the number has more than d significant digits, it is output as if the Gw.d were Ew.d. Numbers output as real are followed by four blanks, and those expressed as exponential are right-justified.

Examples

Field	Stored value	Output value
G14.7	395.94	bbbb395.94bbbb
G1Ø.3	3Ø94.949	bØ.3Ø9EbØ4

H field: wHstring or 'string'

The H descriptor inputs and outputs string data. The string must be exactly the same size as given by w and not enclosed in quotation marks. Blanks are considered characters. Alternatively, you can enclose the string in single quotes, with no size or H listed in the descriptor.

Input: When you use the H field for input, you must put w place holders after the H to reserve space for the incoming value. The new characters replace the place

holders on input. The computer stores the string exactly as input. The same is true if you use quotes.

The next time you use the FORMAT statement, it contains the new literal field. This is convenient, for example, when changing a heading in a FORMAT line.

Examples

Field	Input value	Stored value
8Hbbbbbbbb	bbTRSDOS	bbTRSDOS
'STRING'	TOTALS	TOTALS
'bbbbbbbb'	AVERAGEb	AVERAGEb
6HSTRING	bDOGSb	bDOGSb

Output: Data output with the H FORMAT descriptor is output exactly as listed after the H. The computer outputs data with the H FORMAT descriptor exactly as listed. If you use quotation marks and the output line requires a single quotation mark, then use two successive quotation marks.

Examples

Field	Output
6HbVALUE	bVALUE
8HINCOMEbb	INCOMEbb
'ACCOUNT'	ACCOUNT
'ACC''T NO'	ACC'T NO

Integers

I fields: rIw

I fields input and output integer numbers.

Input: Numbers input under I specifications are right-justified in the field; blanks are considered zeroes.

Examples

Field	Input value	Stored value
I6	-483bb	-483ØØ
I4	bb98	bb98
I8	-4348391	-4348391

Output: Numbers output under I specifications are right-justified in the field. If the number is too large for the field, the computer prints an asterisk before the rightmost w-1 digits.

Examples

Field	Stored value	Output value
I6	-943	bb-943
I4	39849	*849
I8	9Ø93ØØ4Ø	9Ø93ØØ4Ø

Logical

L fields: rLw

The L descriptor inputs and outputs logical data.

Input: Data input under L specifications is either T (TRUE) or F (FALSE). Blanks may precede and follow the pertinent characters.

Examples

Field	Input value	Stored value
L6	bbbTbb	-1
L5	TRUEb	-1
L4	Fbbb	Ø

Output: Data output under L specifications is either T or F with the character being right-justified within its field.

Examples

Field	Stored value	Output value
L1	Ø	F
L3	-1	bbT
L5	Ø	bbbbF

Scaling Factor

P Descriptor: nP field specification

The P descriptor sets the scaling factor on F, G, E, and D input/output so that the computer, depending on the conditions, multiplies or divides the data by the nth power of 1Ø. It automatically sets the scaling factor to zero at the beginning of each formatted READ or WRITE. If it encounters a P descriptor in the FORMAT statement, it changes the scaling factor to the one given. The scale remains the same until a new P is given or the end of the I/O is reached.

Input: During input, the scaling factor causes the input data to be divided by 1Ø**n before being stored. Scaling occurs only on nonexponential values. When the computer inputs exponential data, it ignores the P.

Examples

Field	Input value	Stored value
1PF6.2	775.34	77.534
-2PE1Ø.3	bbbbb4952Ø	4952.Ø
2PF1Ø.3	-Ø.343EbØ4	-34.3ØØ

Output: The effect of the P descriptor on output depends on the type of field following the P. For E, D, and G fields large enough to be considered E fields, the decimal place shifts to the right n times and the exponent is reduced n times (the value remains the same). For F fields, and G fields small enough to be output as F fields, the stored value is multiplied by 1Ø**n before being output.

Examples

Field	Stored value	Output value
1PE12.5	Ø.8842ØEØ5	b8.8842ØEbØ4
2PF6.2	234.91	2349.1
-1PG1Ø.3	4.34	4.34
-1PG1Ø.3	1943.943	bØ.Ø19EbØ5

Skipping Spaces

X field: nX

The X descriptor does not convert any data or refer to any I/O list item. When used, it skips n characters before processing the next field.

Input: The X field causes input to skip over n columns in the record before reading the next data item.

Examples

FORMAT specification	Input value	Stored value
(5X,F6.3)	93844b453Ø1	45.3Ø1
(F5.1,3X,I2)	85.4bbbb1Ø	85.4, 1Ø

Output: The X field causes output to skip n spaces before printing another field.

Examples

FORMAT specification	Stored value	Output value
(5X,F6.3)	-3.342	bbbbb-3.342
(3X,'TOTAL')		bbbTOTAL

Interpreting the Specification List:

For input, the computer reads the specifications from left to right to the end of the list. If it doesn't read all the variables listed in the READ statement, it reuses the FORMAT specifications. Each time it restarts the specification list, the computer reads a new record. It also reads a new record whenever it encounters a slash (/).

Example

```
    READ(5,1Ø) A,B,C,D
1Ø  FORMAT(F6.4,F1Ø.1)
```

In this example, the computer reads A and B from one record and C and D from the next. The format descriptor corresponding to A and C is F6.4, and the descriptor corresponding to B and D is F1Ø.1.

On output, the interpretation follows the same rules, with a few additions. Carriage control statements for your printer and screen are given in the FORMAT specification list. The carriage control character is optional, but if present, it is the first specification in each list.

A zero in that position causes your printer or screen to skip two lines before printing the next record, a 1 causes it to go to the next page before printing, and blank causes it to skip one line. Any other character is not supported by Radio Shack and usually causes the printer to skip one space.

Important Note: On a FORMAT statement for a WRITE to the screen or printer, the first character in the specification list must be carriage control command, FORMAT('Ø',...), a blank character enclosed in quotes, FORMAT(' TEST = ',...), or an X field, FORMAT(5X,...). If it is not one of these, you may lose the first character of each line.

Example

```
      WRITE(5,1Ø) A,B,C,D,E
1Ø    FORMAT('Ø',F6.1)
```

In this example, the computer writes A, B, C, D, and E with a blank line between each.

Slashes also have the effect of a carriage control command on output. For every slash encountered, the computer starts a new line (a new record).

Example

```
      WRITE(5,1Ø) A,B
1Ø    FORMAT(1ØX,F6.2//1ØX,F4.1)
```

In this example, one blank line is between A and B.

You may also use parenthesis inside the specification. A "repeat" factor before a parenthetical expression tells the computer to repeat that particular specification any number of times. For example:

```
FORMAT(2(1ØX,F5.2))
```

is the same as

```
FORMAT(1ØX,F5.2,1ØX,F5.2)
```

READ

The form of the READ statement is:

```
READ(LUN,format label, REC=record number,
END=label, ERR=label) variable list
```

The LUN gives the unit number from which the computer reads data. It is required. The format label refers to a FORMAT statement with that label elsewhere in the program. The format label is optional. If you do not use it, the input is unformatted (unformatted input is only allowed from the diskette). The REC=record number finds the record number of a file by direct access. This is discussed later in this section.

The END=label option tells the computer where to go after reading the last record from a file, and the ERR=label option tells the computer where to go in case of an input errror, such as no file of that name or a hardware problem. If the END= or ERR= options aren't used, and an end-of-file or input error occurs, the statement causes a fatal runtime error (see Error Messages).

The names for storing incoming data make up the variable list. The variables in the list must match in type the corresponding specifications in the FORMAT list. They must also be separated by commas.

You can read in array variables by listing each individual element, by listing only the array name (no subscript), or by using an implied DO loop.

Example 1

```
DIMENSION A(3)
.
.
READ(6,1Ø) A(1),A(2),A(3),B,C
READ(6,1Ø) (A(I),I=1,3),B,C
READ(6,1Ø) A,B,C
```

Each statement READS the three elements of the array A, as well as B and C.

Example 2

```
      BYTE N
      DIMENSION N(2,2,2)
      READ(3,1Ø)(((N(I,J,K),I=1,2),J=1,2),K=1,2)
1Ø    FORMAT(8A1)
      WRITE(3,2Ø)(((N(I,J,K),I=1,2),J=1,2),K=1,2)
2Ø    FORMAT(1X,8A1)
      END
```

This example uses nested implied DO loops to read eight characters from the keyboard into ARRAY N and then write them back to the screen.

The READ line is equivalent to:

```
      DO  1ØØ  I = 1,2
      DO  1ØØ  J = 1,2
      DO  1ØØ  K = 1,2
      READ(3,1Ø) N(I,J,K)
1ØØ   CONTINUE
```

Note that the "I = 1,2" implied DO loop is the innermost and "K = 1,2" is the outermost DO loop.

For two subscripts, the READ statement appears as follows:

```
READ(3,1Ø) ((N(I,J),I = 1,2),J = 1,2)
```

You can also use the READ statement without a variable list. This lets you read Hollerith and literal constants into your FORMAT lines.

Each time your computer executes a READ statement, it skips the remaining portion of the record and reads a new record. (You can also use certain commands in the FORMAT specification list to instruct your computer to read new records. See the discussion on FORMAT earlier in this chapter.)

Example 1

```
      CALL OPEN(6,'ACCT/DAT',15)
5     READ(6,1Ø) ACCTNO,BAL
1Ø     FORMAT(F6.2,2X,F6.2)
       .
       .
       GO TO 5
```

This example opens a file called ACCT/DAT, the records of which are 15 bytes long, and READs the variables ACCTNO and BAL (a two-character space is between ACCTNO and BAL), performs some processing, and then returns to the READ statement, where it reads a new record.

Example 2

```
      DIMENSION PROJCT(25)
      CALL OPEN(7,'LIST/FIL:1',1Ø3)
      READ(7,1Ø) NUM,(PROJCT (I),I=1,25)
1Ø    FORMAT(I2,25F4.2)
      .
      .
      END
```

In this example, you give an array PROJCT the dimension of 25 elements. The computer opens a file named LIST/FIL on Drive 1; its buffer is LUN 7, and each record is 1Ø3 bytes long. The computer reads NUM and then reads the 25 elements of PROJCT.

Example 3

```
      READ(6,ERR=2ØØ) NUM
      .
      .
2ØØ   STOP
```

In this example, you open the file FORTØ6/DAT (a default name) and read an unformatted integer into NUM. The record length is set by default to 128 bytes. Control transfers

to Statement 2ØØ if there is a problem with the input, that is, if no file had that name.

Most of the previous programs are only portions of programs. Several complete programs are at the end of this chapter. But, logically, a program has no use if its only function is to read in data and process it. It must communicate the results of its actions to you or to a disk file. This is done by means of the WRITE statement.

WRITE

The form of the WRITE statement is:

WRITE(LUN,format label, REC=label, ERR=label, END=label) variable list

The LUN specifies the unit number where the data goes. It is required. The format label refers to a FORMAT statement elsewhere in the program. It is optional; if you do not use it, unformatted data is output. (Unformatted data is only allowed for disk file I/O.)

The REC=label option writes to direct access files and refers to the record number. The END=label option gives the label of the statement to which control transfers if an end-of-file is encountered during the WRITE. The ERR=label option gives the label of the statement to which control transfers in case of an output error.

The variable list is optional. If the output is formatted and a variable list is given, the items on the list must have corresponding specifications in the FORMAT list. Each item in the list must be separated by a comma. You can list and array individually as elements, as the array name alone (no subscript), or in an implied DO loop.

If you do not use a variable list, then your computer outputs only the Holleriths and literals in the FORMAT specification. This is useful, for example, for printing headings on tables.

Each time your computer executes a WRITE statement, it writes the data to the proper LUN in its respective field. If, after it writes all the data, record space remains, it fills that space with blanks and begins a new record the next time it executes the WRITE. (You can also start new records with various statements in the FORMAT statement. See the discussion on FORMAT earlier in this chapter.)

If the computer is writing data to a disk where old data exists, it destroys the old data and writes the new data in its place.

Example 1

```
          A = 24.4
          B = 9.4
          C = A + B
          WRITE(5,1Ø) A,B,C
    1Ø    FORMAT(' ',F5.1,1X,F5.1,1X,F5.1)
          END
```

In this example, the computer writes A, B, and C to LUN 5, the screen. They each use a F5.1 field, separated from the neighboring fields by a blank (1X).

Example 2

```
          C = 13.2
          WRITE(7,1Ø) C
    1Ø    FORMAT(F6.2)
          END
```

In this example, the computer writes C to a diskette file of the default name FORTØ7/DAT, using an F6.2 field.

Sequential and Direct Addressing

The two kinds of file addressing in TRS-8Ø FORTRAN are sequential and direct. In the examples so far, we have used

only sequential addressing. In sequential addressing, the computer reads or writes the records in the order that it executes the READ or WRITE statements. Once it accesses a record, it cannot access the same record again without closing and then reopening the file. Direct addressing, on the other hand, lets you access any record of the file at any time.

Both direct and sequential addressing can be advantageous. If you must read or write the whole file, you don't need the record numbers; so you can use sequential addressing. At other times, you may want to update a few selected records and not bother reading over the previous records; so you use direct addressing to save time.

Direct and sequential addressing store information on the diskette in slightly different ways. In sequential addressing, a delimiter (separator) follows each record. In direct addressing, no delimiter exists. A blank record follows the final record.

For example, to store 8 records sequentially in an F6.2 field, you need a record length of 7, and your file is 8 records long. If, on the other hand, you use direct addressing for that file, your record lengths are 6 bytes long, and the file contains 9 records.

Example 1

```
      CALL OPEN(6,'DATA/FIL',11)
      CALL OPEN (7,'UNPAID/FIL',11)
5     READ(6,1Ø,END=3Ø) NUM,AMT
1Ø    FORMAT(I4,F7.2)
      IF (AMT) 2Ø,5,5
2Ø    WRITE(7,1Ø) NUM,AMT
      GO TO 5
3Ø    ENDFILE 6
      ENDFILE 7
      END
```

In this program, your computer reads in records sequentially from DATA/FIL and then writes any records that contain a negative value for NUM sequentially to UNPAID/FIL.

Example 2

```
      CALL OPEN(6,'ACCT/DAT',11)
      INTEGER ACCTNO
      DO 2Ø I=2Ø,3Ø
      READ(6,1Ø,REC=I) ACCTNO,BAL
1Ø    FORMAT(I4,F7.2)
      WRITE(5,15) ACCTNO,BAL
15    FORMAT(' ACCOUNT NUMBER ',I4,(3X,'BALANCE
     X IS',F7.2))
2Ø    CONTINUE
      END
```

In this example, the computer accesses and reads records 2Ø through 3Ø directly from a file call ACCT/DAT. Then it displays these on your screen.

Example 3

```
      REAL NUM(1Ø)
      CALL OPEN(6,'DAT/FIL',7)
      DO 2Ø I=1,1Ø
      READ(6,1Ø) NUM(I)
1Ø    FORMAT(F6.2)
2Ø    NUM(I) = NUM(I) + 2.Ø
      REWIND 6
      DO 3Ø I=1,1Ø
      WRITE(6,1Ø) NUM(I)
3Ø    CONTINUE
      END
```

In this example, your computer reads in a sequential file containing NUM(I). It increments NUM(I) by 2.Ø and then gives the command REWIND. This command closes and then reopens the file under the same specifications as before. Then it writes the updated NUM(I) back to the disk file.

Example 4

```
          DIMENSION A(1Ø)
          CALL OPEN(6,'INVENT/DAT:1',5)
          DO 2Ø I=1,1Ø
          A(I) = FLOAT(I)
          WRITE(6,1Ø,REC=I) A(I)
    1Ø    FORMAT(F4.1)
    2Ø    CONTINUE
          END
```

In this example, a file named INVENT/DAT with 6 byte-long records is opened on the diskette in Drive 1. The elements of the array A are put into successive records of that file, in F4.1 formats.

CHAPTER 9 / SEGMENTING PROGRAMS

FORTRAN, like many other computer languages, runs efficiently as a single main program unit. However, sometimes it is convenient to break the program down into several smaller subprograms. This is called segmenting the program.

Segmenting a program may serve several purposes. First, it can reduce the main program to a few key commands, thus making it easier to discern and understand.

Segmenting is sometimes necessary when more than one person works on a program. By assigning a certain segment of the program to one person and another part to a second person, you can write the entire program easier and quicker. Segmenting assures a greater compatibility between each person's coding.

Segmenting can be very helpful if parts of your program are long or require great amounts of storage and other portions of the program are short and require little storage. You may use the long portion only occasionally but still want it available. If you make the long program a subprogram, you have access to it because the L8Ø linker lets you link a main program with any compiled subprogram on your diskette.

You can segment your programs by using a series of GO TOs, or you can use special FORTRAN features known as subprograms. Subprograms are program modules that lie before or after the main program and that the main program can access. The main program transfers control to the subprogram, executes the subprogram statements, and transfers control back to the main program.

FUNCTIONs

FUNCTIONs are subprograms that process data given them by the main program; they then return one value. You may already be familiar with one type of FUNCTION, the intrinsic library function, which finds, for example, square root (SQRT) and trigonometric functions (for example, COS).

FORTRAN lets you create your own functions that you can access just like a library function. You may call a function from your main program by listing it as an expression in a statement line, with the parameters (variables and/or constants) it is to use listed parenthetically after it. For example:

```
TOT = D + AVE(A,B,12.6)
```

says that TOT is to be set equal to the sum of D and a value found by a FUNCTION called AVE. AVE uses the parameters A, B, and 12.6 in its statements.

The basic syntax of a FUNCTION subprogram is:

```
FUNCTION function name(local variables)
.
function name = expression
RETURN
END
```

For example, you can use the AVE mentioned above to average A, B, and 12.6. The subprogram takes the form:

```
FUNCTION AVE(X,Y,Z)
W = X + Y + Z
AVE = W / 3
RETURN
END
```

Note that the main program statement used A, B, and 12.6 for the parameter names, but in the FUNCTION, we used X, Y, and Z. The latter variables are known as "local variables."

You use local variables to express the main program parameters in the subprogram. By using local variables, you can access the function several times using different sets of parameters each time. For example, both

```
SAL = AVE(24.9,B,C)
TOT = AVE(D,E,F)
```

may access the function AVE.

Note: You may want to use the same main program variable names to represent the local variable in the subprogram. This is perfectly legitimate; however, in this chapter you use variables with differing local variables.

Consider this short program:

```
      DIMENSION A(1Ø)
      REAL MEAN
      N = Ø
1     READ(5,5) A(N)
5     FORMAT(F6.2)
      IF(A(N).EQ.Ø.Ø) GO TO 1Ø
      N = N + 1
      GO TO 1
1Ø    MEAN = AVE(A,N)
      WRITE(5,2Ø) MEAN
2Ø    FORMAT(' AVERAGE = ',F6.2)
      END


      FUNCTION AVE(X,I)
      DIMENSION X(1Ø)
      DO 1Ø J=1,I
1Ø    TOT = TOT + X(J)
      AVE = TOT / I
      RETURN
      END
```

This program demonstrates several rules concerning subprograms. The main program parameters that are passed to

the function are the real array A and the integer N. Their counterparts in the function are X and I. Note that the local variables must agree in type with those listed in the main program and that you must declare the size of the local variable X. Furthermore, we use the same statement label in the subprogram (label 1Ø) that we use in the main program. You may do this since the two program units are separate. Finally, note how the name of the function (in this case AVE) is the variable name of the final result of the function.

SUBROUTINEs

SUBROUTINEs are another type of subprogram much like FUNCTIONs. However, unlike FUNCTIONs, the SUBROUTINE can return any number of parameters. For example, in the program above, it returns the value AVE. A SUBROUTINE, on the other hand, may find several values and return all of them.

Unlike functions that you list as expressions in the program, you access SUBROUTINEs from the main program by means of a CALL statement. The CALL statement consists of the word CALL, the name of the subroutine, and the parameters (constants and/or variables) listed parenthetically. For example:

```
CALL SUB1(A,B,12.6)
```

SUB 1 is the name of the subroutine, and A,B, and 12.6 are the parameters passed to and from the subroutine. An important difference here is that the variables in the parameter list may or may not be defined previously in the program. For instance, in the CALL statement above, A might have been defined in the main program, and B is to be found by the subroutine.

The subroutine itself has the following syntax:

```
      SUBROUTINE subroutine name(local variables)
      .
      .
      RETURN
      END
```

Consider this short program:

```
      PROGRAM MAIN
      READ(5,1Ø) A,B,C
1Ø    FORMAT(F6.1)
      CALL PROCSS(A,B,C,TOT,AVE)
      WRITE(5,2Ø) A,B,C,TOT,AVE
2Ø    FORMAT(' A = ',F6.1/' B = ',F6.1/' C = ',F6.1/
     1 ' TOTAL = ',F6.1/' AVERAGE = ',F6.1)
      END

      SUBROUTINE PROCSS(X,Y,Z,SUM,MEAN)
      REAL MEAN
      SUM = X + Y + Z
      MEAN = SUM / 3.Ø
      RETURN
      END
```

In this example, A, B, and C are read in from the main program, and the subroutine returns the sum and average. The variables TOT and AVE are undefined until the subroutine is CALLed and executed. Again note that the parameters of the main program must match the subroutine local variables in type and number.

The main program and the subroutine may pass parameters back and forth, as shown above, or the passing may be only one way. For example, if you want a subroutine to read in data, it is not necessary to send the main program variables to the subroutine. For example:

```
      CALL READER(A,B,C)
      .
      .
      END
      SUBROUTINE READER(X,Y,Z)
      READ(5,5) X,Y,Z
5     FORMAT(F6.2)
      RETURN
      END
```

In this example, A, B, and C have no values until READER is executed.

It is also possible to have a program in which no parameters are passed. For instance, if you wanted a program to print all your column headings, you might have:

```
      CALL HEADER
      .
      .
      END
      SUBROUTINE HEADER
      WRITE(2,1Ø)
1Ø    FORMAT('1',3ØX,'HOURLY REPORT OF'/32X,'TEMP'
     1  'EXPERIMENT'///25X,'METER',2ØX,'CURRENT'
     2  25X, 'NUMBER',19X,'READING')
      RETURN
      END
```

MORE SUBROUTINE OPTIONS

It is possible in F8Ø FORTRAN to pass parameters by means of COMMON statements. COMMON statements assign parameters in the main program to the same memory location as the local

variables of the subroutine. The necessary main program parameters are put into a "common block" of storage in the main program, somewhere before the CALL to the subroutine. Inside the subroutine, another COMMON statement declares the local variables to be in the same memory location as the actual parameters. For example:

```
COMMON A,B,I
CALL SUB1
.
.
END
SUBROUTINE SUB1
COMMON X,Y,J
.
.
END
```

In this example, A, B, and I are put into the same memory location as X, Y, and J. A is in the same location as D, B is in the same location as E, and I is in the same location as J. The advantage of using COMMON instead of listing the variables after the CALL and SUBROUTINE statements is that you save memory space. The above lines are equivalent to:

```
CALL SUB1(A,B,I)
.
END
SUBROUTINE SUB1(X,Y,J)
.
.
END
```

You can name the COMMON memory location blocks so that you can use several different blocks without interfering with the data in other blocks. For example:

```
COMMON/AREA1/A,B,C/AREA2/D,E,F
CALL SUB1
CALL SUB2
.
.
END
SUBROUTINE SUB1
COMMON/AREA1/T,U,V
.
.
END
SUBROUTINE SUB2
COMMON/AREA 2/X,Y,Z
.
.
END
```

In the above example, the block called AREA1 contains A, B, and C, and their SUB1 counterparts T, U, and V. AREA2 contains D, E, and F, and their SUB2 counterparts X, Y, and Z.

To use variables in COMMON storage and to initialize them with DATA statements, you must use a subprogram called BLOCK DATA. As you can with the other subprograms, you can place BLOCK DATA either before or after the main program. Its syntax is:

```
BLOCK DATA subprogram name
COMMON/common block names/elements/
common block names/elements
DATA variable/data list
END
```

An example of a BLOCK DATA subroutine is

```
BLOCK DATA PRG1
REAL ITEM(3)
COMMON/AREA1/A,B/AREA2/ITEM
DATA A/34.3/ITEM/12.3,12.6,12.9/
END
```

Note the following rules demonstrated by this program:

- You must declare the size of any arrays used in BLOCK DATA subprograms.
- It is not necessary to assign a value to every variable in the COMMON list; however, you must list every item in the DATA list in the COMMON list. In other words, you may only initialize COMMON data in BLOCK DATA subprograms.
- The BLOCK DATA subprogram cannot contain any executable statements.

Using EQUIVALENCE and COMMON Together

You can use COMMON statements in conjunction with EQUIVALENCE statements. Listing an EQUIVALENCE statement after a COMMON statement causes the equivalent variables to be put into the COMMON storage block. For example:

```
COMMON A,B,C
EQUIVALENCE (A,D)
```

in effect puts D into COMMON storage, since it is equivalent to A.

You may extend the size of the COMMON area by using arrays in the EQUIVALENCE statement. For example:

```
DIMENSION A(1Ø)
COMMON B,C
EQUIVALENCE (A(1),B)
```

This statement stores A(1) and B in the same location and in a COMMON block. It also makes A (2) equivalent to C and extends the size of the COMMON block to include the remaining members of A.

Note: You can extend COMMON storage forward but not backward. For example:

```
COMMON B,C
EQUIVALENCE (A(2),C)
```

is legal. It makes A(1) and B equivalent and A(2) and C equivalent. However,

```
COMMON B,C
EQUIVALENCE (A(3),C)
```

is not legal because it implies that A(2) is equivalent to B and that A(1) will be placed before B in the COMMON block.

You can list subprograms in the same file as the main program. All examples so far have done that. However, you may insert the subprogram into the object code in two ways.

INCLUDE

The INCLUDE statement causes a FORTRAN source file on your diskette to be included in your program at the location in the program where the INCLUDE statement is listed. Its syntax is:

```
INCLUDE filename
```

Since your subprograms can be either before or after the main program, the INCLUDE statement should be either before or after the main program. For example:

```
PROGRAM SUMS
.
.
CALL PRINTR
.
END
INCLUDE PRT/FOR:1
```

This program CALLs a subroutine named PRINTR. This subroutine is stored in a file called PRT/FOR on Drive 1. If you compile this program creating a listing file, you

find that this subroutine has been INCLUDEd after the main program.

Linking the Subprogram

Linking subroutines to the main program while in the L8Ø command mode is another way to use subroutines that your main program accesses. For example:

```
L8Ø MAIN-N,SUB1,SUB2,FUNC1,MAIN-E
```

links the subprograms SUB1, SUB2, and FUNC1 to the main program MAIN. The subprograms must, of course, be compiled just as the main program was. (For further information, see Chapter 4, "The Linker.")

```
C     The object of this program is to find the average
C     and standard deviation for a list of numbers that
C     you input.  This program uses subroutines and
C     functions.  The main program CALLs READER, the
C     subroutine by which you input your list, a
C     subroutine to record the list on a diskette
C     called DSKRIT.  It then uses two user-programmed
C     functions, AVE an STD, which find the average
C     and standard deviations respectively, and finally
C     it CALLs a subroutine which outputs the results to
C     the screen.  Note that each subroutine uses the
C     list stored in an array, and in each subprogram
C     it must be dimensioned.  Also note that the
C     subroutines all use local variables in their
C     variable list.  This is not required, but is often
C     helpful in distinguishing between main- and
C     subprogram variables.

      PROGRAM CALLER
      DIMENSION RESULT (1Ø)
      CALL READER(RESULT,N)
      CALL DSKRIT(RESULT,N)
      AVERAG = AVE(RESULT,N)
```

```
      STDDEV = STD(RESULT,AVERAG,N)
      CALL WRITER(RESULT,N,AVERAG,STDDEV)
      END

C     This is the subroutine for inputting the list,
C     and is called READER

      SUBROUTINE READER(ARRYIN,I)
      DIMENSION ARRYIN(1Ø)
      DO 3Ø I=1,1Ø
      WRITE(5,1Ø) I
   1Ø FORMAT(' RESULT #',I3,' IS: ')
      READ(5,2Ø) ARRYIN(I)
   2Ø FORMAT(F8.3)
      IF(ARRYIN(I).EQ.Ø.Ø) GO TO 4Ø
   3Ø CONTINUE
      RETURN
   4Ø I = I - 1
      RETURN
      END

C     This subroutine, called DSKRIT, writes the list
C     to a diskette file called RESULT/DAT.

      SUBROUTINE DSKRIT(ARYOUT,I)
      DIMENSION ARYOUT(1Ø)
      CALL OPEN(6,'RESULT/DAT',15)
      DO 2Ø J=1,I
      WRITE(6,1Ø) ARYOUT(J)
   1Ø FORMAT(F8.3)
   2Ø CONTINUE
      RETURN
      END

C     This subroutine finds the average of the list, and
C     is called AVE.

      FUNCTION AVE(ARRY,I)
      DIMENSION ARRY(1Ø)
      DATA SUM/Ø.Ø/
      DO 1Ø J=1,I
```

```
   1Ø SUM = SUM + ARRY(J)
      AVE = SUM / I
      RETURN
      END

C     This function finds the standard deviation of
C     the list, and is called STD.

      FUNCTION STD(ARRY,AVRG,I)
      DIMENSION ARRY(1Ø)
      DATA SUM/Ø.Ø/
      DO 1Ø J=1,I
   1Ø SUM = SUM + (ARRY(J) - AVRG) **2
      STD = SQRT(SUM / I)
      RETURN
      END

C     This subroutine, called WRITER, outputs the list
C     its average, and its standard deviation to the
C     screen.

      SUBROUTINE WRITER(ARYOUT,I,AVE,STD)
      DIMENSION ARYOUT(1Ø)
      WRITE(5,5)
    5 FORMAT (' THE LIST OF RESULTS IS')
      DO 1Ø J=1,I
   1Ø WRITE (5,2Ø) J,ARYOUT(J)
   2Ø FORMAT(' RESULT #',I3,'  IS: ',F8.3)
      WRITE(5,3Ø) AVE,STD
   3Ø FORMAT( ' THE AVERAGE IS: ',F8.3/
     *       ' THE STANDARD DEVIATION IS: ',F8.4)
      RETURN
      END
```

CHAPTER 1Ø / FORTRAN STATEMENTS

FORTRAN statements tell the Computer to perform an operation. This chapter contains an alphabetical listing of each statement, with a brief definition of what the statement does. Each listing includes:

1. The type of statement (executable or nonexecutable). This is noted at the top right corner.
2. A reference to other chapters that discuss the statement (if any). This is also noted in the top right corner.
3. The syntax to use in typing the statement. This is in the gray box.
4. A description of the syntax.
5. Examples.

Three statements -- OPEN, OUT, and POKE -- are actually subroutines. Therefore, you use them with the CALL statement. For example:

```
CALL OUT(6,A)
```

CALLs the subroutine OUT.

ASSIGN Executable
Setting values for ASSIGNed GO TOs

ASSIGN integer constant TO integer variable

Gives a value to variables that the computer uses in an ASSIGNed GO TO. The integer constant must be a statement label, and the integer variable, the name of the variable used in the ASSIGNed GO TO.

Example

```
ASSIGN 10Ø TO LABEL
```

sets label equal to 10Ø. Presumably an ASSIGNed GO TO follows this statement later in the program.

See the Assigned GO TO statement for additional information on how to use this statement.

BLOCK DATA Nonexecutable
Titling BLOCK DATA subprograms Chapter 9

BLOCK DATA subprogram name

Titles block data subprograms that initialize common block data. The BLOCK DATA statement must be the first statement of the block data subprogram. The subprogram name may be from one to six alphanumeric characters, the first of which must be a letter.

Example

```
BLOCK DATA SUBPRG
```

defines the block data subprogram SUBPRG.

BYTE Nonexecutable
Declaring a byte variable Chapter 6

BYTE variable name(dimension)variable name(dimension)

Declares the given variables to byte size variables. For arrays, you can give the dimension of the array in the statement.

Example

```
BYTE CODE,NAME(2Ø)
```

declares CODE to be a byte variable and NAME to be a byte array consisting of 2Ø elements.

CALL
Accessing a subroutine.

Executable
Chapter 9

CALL subroutine name(parameters)

Accesses a subroutine either from your program or from the FORTRAN FORLIB library. The subroutine name can be from one to six alphanumeric characters, the first of which must be a letter. The parameters are optional. If used, they must match in type and number with the local variable list in the SUBROUTINE statement of the subroutine.

Example

```
CALL READ1(A,B,N)
```

calls a subroutine by the name of READ1, which uses the variables A, B, and N.

If the subroutine is in assembly language, subroutine name must be a PUBLIC symbol. (See Appendix M8Ø).

COMMON Nonexecutable
Common memory location declaration Chapter 9

```
     COMMON variable list
or   COMMON/block area name/variable list...
```

Assigns variables to COMMON areas of storage in order to pass parameters between the main program and subroutines. The block area name is optional. If used, it must consist of one to six alphanumeric characters, the first of which is a letter. The order of the variable list of the main program COMMON statement must be consistent in type with the list of the COMMON statement in the subroutine.

Example 1

```
COMMON A,B,I
```

defines A, B, and I to be in common storage.

Example 2

```
COMMON/AREA1/A,B,C/AREA2/D,E,F
```

defines A, B, and C to be in a common block called AREA1, and D, E, and F to be in a block called AREA2.

CONTINUE
A "No-operation" executable statement

Executable
Chapter 5

```
CONTINUE
```

Performs no actual operation but serves as a place to jump to when an executable statement is required. For instance, CONTINUE most commonly signifies the last line of a DO loop, since DO loops are required to end on an executable statement. (For more information, see DO Loops.)

Example

```
     DO 1Ø I=1,5
     .
     .
1Ø CONTINUE
```

The final statement in the loop is the CONTINUE.

DATA — Nonexecutable
Initializing variables — Chapter 6

DATA variable list/data list/
variable list/data list/.....

Initializes variables at the beginning of the program. The items in the variable list are separated by commas and must have corresponding data in the data list. The variable list and the data list must match in type. You can list arrays by the variable name without any subscript.

Example 1

```
DATA A,B,N/12.4,14.5,9/
```

sets A equal to 12.4, B equal to 14.5, and N equal to 9

Example 2

```
DIMENSION N(1Ø)
DATA N/1Ø*Ø/B,C/Ø.Ø,Ø.Ø/
```

sets the ten elements of N equal to Ø, and B and C equal to Ø.Ø. Note that the data list can be listed as:

number of items * initial value

DECODE Executable
Converts string to numeric.

DECODE (array,format statement label)variable list

Converts the contents of an array of characters into a number and stores it in variable list. The resulting number is in the format specified by the format statement.

When decoded into a logical variable, a "T" or "F" in the array is decoded as true or false (Ø or -1).

Note to BASIC programmers: DECODE is similar to the VAL statement.

Example 1

```
DECODE (A,1ØØ)B
```

converts the string in array A into a number and stores it in B. FORMAT Line 1ØØ is used.

Example 2

```
      DIMENSION I(1)
      I(1)='12'
      DECODE (I,2) J
2     FORMAT(I2)
      WRITE(5,3) J
3     FORMAT(1X,'STRING CHARACTERS 12 ARE NOW DECODED
     X AS NUMERIC : ',I2)
      END
```

decodes the string contained in I('12') into a number and stores it in J. The resulting number is formatted by Line 2.

Example 3

```
          DOUBLE PRECISION J
          BYTE I(14)
          WRITE(5,1)
    1     FORMAT(1X,'TYPE A 14 DIGIT DECIMAL NUMBER ; ')
          READ (5,2) I
    2     FORMAT(14A1)
          DECODE(I,3) J
    3     FORMAT(F2Ø.13)
          WRITE (5,4) J
    4     FORMAT(1X,'NOW IT IS DECODED INTO THIS NUMBER :
   X ',F2Ø.13)
          END
```

lets you type in a 14-digit number that is stored as a string in array I. It is then decoded into J, using the F2Ø.13 format.

Example 4

```
          PROGRAM DEC
          BYTE A(12)
          DATA A/'1','5','.','2',7',' ','1','6','3','.',
        X '2','4'/
          WRITE(3,99) A
   . 99   FORMAT (1X,12A1)
          DECODE(A,1ØØ) B1,B2
    1ØØ   FORMAT(2F6.2)
          WRITE(3,1Ø1) B1,B2
    1Ø1   FORMAT(1X,2F8.2)
          END
```

decodes array A into two variables, B1 and B2, using format line 1ØØ. Notice that Line 1ØØ specifies a field wide enough to hold the widest number (163.24). Also notice that the two numbers in array A are separated by a blank character (' ').

When using DECODE, do not use an array with leading spaces, or DECODE will assume that the array is empty.

DIMENSION Nonexecutable
Dimensioning a variable Chapter 6

DIMENSION array name(dimensions),
array name (dimensions),...

Reserves memory space for arrays. The array may have a maximum of three dimensions, and the total size of the array depends on the size of available memory. The statement must precede all executable program statements.

Examples

```
DIMENSION A(1Ø)
```

reserves 1Ø storage locations for A.

```
DIMENSION A(1Ø,1Ø),B(5),C(3,3,4)
```

reserves 1ØØ locations (1Ø x 1Ø) for the array A, 5 for B, and 36 (3 x 3 x 4) for C.

DO
Looping

Executable
Chapter 5

DO statement label integer variable=starting value,
ending value,increment

Defines a "DO loop." The statement label represents the lower boundary of the loop. The integer variable initially equals the starting value and increases by the value of the increment each time the loop is executed.

The DO statement repeatedly executes the loop to the labeled statement until the integer variable is greater than the ending value. The starting value must be a positive integer constant or variable, and the ending value must be an integer constant or variable greater than or equal to the starting value. The increment is a positive integer constant or variable and is optional. If not listed, the increment is 1.

The following rules apply to DO loops:

The integer variable may not be an extended integer. You may use it as a variable in the loop (for instance, as a subscript), but it must not be modified inside the loop.

The final statement of the loop must be an executable statement other than GO TO, RETURN, STOP, PAUSE, an arithmetic IF, or another DO loop. If a logical IF is the final statement, it may not have one of the prior statements as its "true" option. A common way of ending a DO loop is with a CONTINUE statement.

You may nest DO loops inside other DO loops; however, they must be completely inside the next loop or terminate on the same statement. For example:

```
        DO 1Ø I=1,1Ø
        .
        DO 5 J=1,1Ø
        .
5       CONTINUE
1Ø      CONTINUE

        valid
```

```
        DO 1Ø I=1,1Ø
        .
        DO 5 J=1,1Ø
        .
1Ø      CONTINUE
5       CONTINUE

        invalid
```

Example 1

```
        DO 1Ø I=1,1Ø
        .
        .
        .
1Ø      CONTINUE
```

The loop runs from the DO statement to the CONTINUE statement. The loop is executed 1Ø times (I is equal to 1,2,3....1Ø).

Example 2

```
        DO 1Ø I=1,1Ø
        .
        DO 1Ø J=1,1Ø
        .
        DO 5 K=1,2Ø,2
        .
5       SUM = SUM + X(K)
        .
1Ø      CONTINUE
```

One loop is nested inside another loop that is nested inside another loop. Two of the loops end with the same CONTINUE statement; the innermost ends with a replacement statement. The innermost loop is executed 1Ø times (K = 1,3,5,...19), and the other two are also executed 1Ø times (I (and J) = 1,2,3...1Ø).

DOUBLE PRECISION Nonexecutable
Declaring a variable to be double precision Chapter 6

DOUBLE PRECISION variable(dimensions),
variable(dimensions),...

declares the listed variables to be double precision variables. If the variable represents an array, you can also define its size in the statement.

Example

```
DOUBLE PRECISION DTOT,LIST(1Ø)
```

declares DTOT to be double precision and LIST to be a double precision array.

IMPORTANT NOTE: If you do not use the D notation when typing a double precision number, FORTRAN assumes it is a real number and rounds it to seven significant digits. (For further information, see Chapter 6, "DATA.")

ENCODE Executable
Changing internal format Chapter 5

ENCODE(array,format statement label)variable list

Converts the numeric contents in the variable list into a string and stores it in the array. The format statement controls the format of the resulting array.

Note to BASIC programmers: ENCODE is similar to the STR$ statement.

Example 1

```
ENCODE(A,1ØØ) B,C,D,E
```

converts B, C, D, and E into strings and stores them into array A using the format line 1ØØ.

Example 2

```
      BYTE L(2Ø)
      R1 = 45.62
      R2 = 5Ø.ØØ
      ENCODE(L,1ØØ) R1,R2
1ØØ   FORMAT(2F5.2)
      WRITE(5,2ØØ)L
2ØØ   FORMAT(1X,2ØA1)
```

formats 45.62 and 5Ø.Ø into the 2F5.2 format, converts them into strings, and stores them in string array L.

Note: Logical variables are encoded as "T" for TRUE and "F" for FALSE.

END
Termination of the program

Executable
Chapter 5

```
END
```

The physical termination of the program. This statement must be the last statement in a main program or subprogram.

Example

```
A = 12.4
WRITE(6) A
END
```

The END statement is the last statement in this short program.

ENDFILE
Closing a file

Executable
Chapter 8

ENDFILE logical unit number

Closes an open file. The logical unit number (LUN) must be an integer constant or variable.

Example

```
ENDFILE 6
```

closes the file accessed by LUN 6.

EQUIVALENCE Nonexecutable
Declaring equivalent memory locations Chapter 6

EQUIVALENCE (variable list),(variable list),...

Lets you use the same memory location for several variables during program execution. This gives more memory space for the rest of the program. All variables inside each set of parentheses are in the same memory location.

Examples

```
EQUIVALENCE (A,B,C)
```

sets A, B, and C into the same memory location.

```
DIMENSION A(1Ø),B(1Ø)
EQUIVALENCE (A(1),B(1))
```

sets the elements of A in the same memory locations as the elements of B. That is, A(1) and B(1) are in the same spots.

```
DIMENSION A(1Ø),B(5)
EQUIVALENCE (A(6),B(1))
```

sets A(6) in the same location as B(1), A(7) with B(2), and so on.

```
DIMENSION A(1Ø),B(2,5)
EQUIVALENCE (A(1Ø),B(1Ø))
```

sets the elements of A into the same locations as the elements of B. (**Note:** You can describe arrays in the EQUIVALENCE statement by their size alone. In the above example, B is 2 X 5 array (1Ø elements), and for EQUIVALENCE statements, B(2,5) and B(1Ø) have the same meaning as do B(1,1) and B(1).)

```
EQUIVALENCE (A,B,C,D),(E,F)
```

sets A, B, C, and D into the same location, and E and F in another location.

```
DIMENSION D(5)
COMMON A,B,C
EQUIVALENCE (C,D(1))
```

sets C and D into the same location, and also extends the COMMON area to include all of D (the total COMMON area is now seven words long).

```
DIMENSION D(5)
COMMON A,B
EQUIVALENCE (B,D(2))
```

sets B and D(2) into the same location and sets A and D(1) into the same location. It also extends the COMMON area to include the remainder of D. The common area is:

```
A and D(1)
B and D(2)
D(3)
D(4)
D(5)
```

Note: If you are using subscripted (array) variable and a scalar variable that have the same length, you must specify the subscript of the array element. For example, if A is a 4-byte real number and B(1) is a 4-byte array element:

```
EQUIVALENCE(A,B)
```

is not valid, whereas:

```
EQUIVALENCE(A,B(1))
```

is valid.

EXTERNAL Nonexecutable
Using functions inside of functions Chapter 9

EXTERNAL subprogram name list

You must use the EXTERNAL statement when an existing subprogram is part of the parameter list of another subprogram. The subprogram name represents the name of either a library subprogram or a subprogram defined in the program. The EXTERNAL statement must precede any executable statements.

Example

```
EXTERNAL SQRT
A = FUNC(A1,A2,SQRT)
.
.
END
FUNCTION FUNC(A,B,C)
FUNC = C(A/B)
RETURN
END
```

In this example, the SQRT function is listed as a parameter in the functin FUNC where its "local" variable name is C; so it must be listed as EXTERNAL.

FORMAT Nonexecutable
Formatted I/O Chapter 8

label FORMAT(specification list)

Describes to the computer the type and physical appearance of I/O data. The label is a statement label that corresponds to the statement label referenced in a READ or WRITE. The specification list consists of the following:

Carriage Control Characters

The carriage control character functions only with output to the printer or to the screen. When used, it is the first specification in the list and may be

'0'	or	1H0	skip 2 lines before printing
'1'	or	1H1	skip to the top of the next page before printing
anything else			skip one line before printing

The computer never prints the carriage control character. If you do not use it in a WRITE to the printer or screen, the computer takes the first character of the next specificiation as the carriage control character and thus does not print it. This may result in an error if that specification is to contain printed data.

Field Descriptors

The field descriptors describe the actual appearance of the data. Any number of descriptors may appear in the specification list. The descriptors are:

rAw	character I/O
rDw.d	double precision exponential I/O
rEw.d	exponential I/O
rFw.d	floating point I/O
rGw.d	general I/O
rHstring	Hollerith (literal) I/O
rIrIw	interger I/O
rLw	logical I/O
rPfield	scaling I/O fields
wX	blank I/O
'string'	literla (Hollerith) I/O

where r is the number of times to repeat the field, w is the total width of the field, and d is the width of the decimal portion.

Field Separators

Commas or slashes separate individual field descriptors. Commas serve only to separate the fields, but slashes also serve as record terminators. Each time the computer encounters a slash, it ends the current record and considers the next field a new record.

Multiple slashes imply that the computer skip several records before processing more data -- each slash represents one record. For data it is writing to the screen or printer, this means that the computer begins a new line whenever it encounters a slash.

Parentheses

As shown in the syntax, parentheses enclose the entire specification list. You may also include parentheses as a specification. For example, parentheses may enclose several field descriptors, and an integer may precede them.

```
4(F5.1,2X,I2)
```

This is analogous to the r in the field descriptors. It means that the computer uses the enclosed list of descriptors four times before going to the next descriptor.

Interpretation of the Specification List

The computer interprets the FORMAT list from left to right. Each I/O item must have a corresponding field descriptor. If the computer uses all the field descriptors before it inputs or outputs all the I/O items, it repeats the entire specification list. This constitutes a new record.

You may also use FORMAT statements with ENCODE/DECODE commands (see ENCODE/DECODE).

Example 1

```
10 FORMAT (I5,E16.9)
```

describes a record with the following layout:

```
i i i i i s Ø . d d d d d d d d d E s e e
- - - - - - - - - - - - - - - - - - - - -
```

where i represents the integer number, s represents the sign of the exponential number and of the exponent, d represents the decimal portion of the exponential number, and e represents the exponent. Any of the i's may be used for the sign of the integer as needed.

Example 2

```
10 FORMAT ('1',10X,'ITEM NOS',3(3X,I4))
```

describes a record with the following layout:

```
bbbbbbbbbbITEMbNOSbbbiiiibbbiiiibbbiiii
```

where the b's are blanks and the i's represent the integer numbers.

In addition, the record prints on a new page because of the '1'.

Other examples of valid formats:

```
10 FORMAT (1X,F6.2,2X,F6.1)

20 FORMAT (10I4,D20.13)

75 FORMAT (30X,'TOTALS'//(30X,G16.9))
```

FUNCTION
Defining a function

Nonexecutable
Chapter 9

FUNCTION function name(local variable list)

Defines a function subprogram unit and is the first statement of a function subprogram. The function name represents the variable found by the function, and it represents the variable used to call the function.

The local variable list should correspond in type and number with the parameters used in the main program function call.

Example

```
FUNCTION AVERG(A,B,C)
```

titles a function called AVERG and uses the local variables A, B, and C.

GO TO
Unconditional GO TO's

Executable
Chapter 5

GO TO statement label

Instructs your computer to move to another part of the program and to begin execution there rather than following the physical order of the program.

The statement that a GO TO jumps to must be executable. When an unconditional GO TO is encountered in a program, control immediately transfers to the statement with the given label. The label must be an integer number.

Example 1

```
          GO TO 1ØØ
          .
          .
    1ØØ   CONTINUE
```

transfers control to Statement 1ØØ.

Example 2

```
          IF (A.LT.B) GO TO 1ØØ
          A = B
    1ØØ   C = A + D
```

The GO TO is used in conjunction with a logical IF statement. If the logical statement is true, control transfers to Statement 1ØØ; if the statement is false, the A = B statement is executed and then Statement 1ØØ.

GO TO
Computed GO TOs

Executable
Chapter 5

GO TO (statement label list), integer variable

Transfers control to the nth label in the statement label list where n is the integer variable. The variable must be positive, not an extended integer, and its value must have been set before execution of the GO TO statement. If the variable is less than 1 or greater than the number of statement labels, control passes to the statement following the GO TO.

Example

```
J = 3
.
.
GO TO (1Ø,2Ø,3Ø,4Ø,5Ø), J
```

sets J equal to 3. When the GO TO is executed, the computer finds the third label in the list, which is 3Ø, and transfers control to that label.

GO TO
Assigned GO TOs

Executable
Chapter 5

GO TO integer variable,(statement label list)

Transfers control to the labeled statement in the list that is equal to the integer variable. The variable cannot be an extended integer and must have received its value from an ASSIGN statement (see ASSIGN). The statement label list is optional.

Example

```
ASSIGN 1ØØ TO LABEL
.
.
GO TO LABEL,(1ØØ,2ØØ,3ØØ)
```

gives LABEL the value 1ØØ, and when the GO TO is executed, control goes to 1ØØ.

IF
Arithmetic IF

Executable
Chapter 7

IF (expression) label-1,label-2,label-3

Evaluates an expression and, based on the result, decides what action to take next. The expression can be arithmetic, logical, or relational.

The arithmetic IF statement evaluates the expression in parentheses, and if the value is negative, control is passed to label-1. If it is zero, control passes to label-2, and if it is positive, control passes to label-3. Any labels may be identical if necessary.

Example 1

```
IF(I-J) 1ØØ,2ØØ,3ØØ
```

If I-J is negative, the program goes to 1ØØ; if Ø, to 2ØØ; and if positive, to 3ØØ.

Example 2

```
IF (K) 1ØØ,1ØØ,2ØØ
```

If K is less than or equal to Ø, control passes to 1ØØ; otherwise, control passes to 2ØØ.

IF
Logical IF

Executable
Chapter 7

IF(logical expression) executable statement

Evaluates the logical expression in parentheses, and if it is a "true" expression, executes the statement. If the expression is "false," the program passes on to the next statement. The statement can be any executable statement except a DO statement or another logical IF.

Example 1

```
IF ( A.LT.B) B=(B-A)/B
C = SQRT(B)
```

compares A to B. If A is less than B, then the statement B=(B-A)/B is executed, followed by the C=SQRT(B) statement. If A is greater than or equal to B, then the B=(B-A)/B statement is ignored and the C = SQRT(B) statement is executed.

Example 2

```
IF (A-12.Ø.LT.B.AND.B.LT.C) CALL SUB1(A,B,C,D)
D = A
```

calls the subroutine if both A-12.Ø.LT.B and B.LT.C are true. If they are not true, D=A is executed immediately.

Example 3

```
IF (L1.OR.L2) GO TO 2Ø
```

L1 and L2 are logical variables. If either L1 or L2 is true, control passes to 2Ø.

IMPLICIT
Declaring a range of default variable types

Nonexecutable
Chapter 6

IMPLICIT type(range),type(range)...

Redefines the default variable types. Type can be BYTE, REAL, INTEGER, DOUBLE PRECISION, LOGICAL, or INTEGER*4. The range can be one letter or a range of letters represented by

first letter - last letter.

When the computer encounters a variable that starts with that letter, it considers it the type given to it in the IMPLICIT statement.

Examples

```
IMPLICIT REAL(I-N),INTEGER(A-H,O-Z)
```

declares variables starting with the letters I through N to be real and those beginning with A through H and O through Z to be integers (exactly opposite of the default values).

```
IMPLICIT DOUBLE PRECISION (D)
```

declares all variables starting with D to be double precision variables.

```
IMPLICIT INTEGER*4(I-N)
INTEGER ITEM,NUMBER(1Ø)
```

declares all variables starting with the letters I-N to be extended integers except for ITEM and NUMBER.

INCLUDE Nonexecutable
Bringing in outside programs Chapter 9

INCLUDE filename

Brings an outside source code into the current program at the location in the main program where the INCLUDE statement is listed. The INCLUDEd code can be, for example, a commonly used subroutine or function. filename is any valid TRSDOS file name. (Note: The included program can contain only one program or subprogram; that is, the program can have only one END statement.)

Example

```
INCLUDE PRINT/FOR
.
.
.
CALL PRINTR(NUM,DEP,BAL)
.
.
```

brings a file called PRINT/FOR into the main program. This file contains a subroutine called PRINTR that the computer later CALLs.

INTEGER Nonexecutable
Declaring a variable to be integer. Chapter 6

INTEGER variable(dimensions),variable(dimensions),..

Declares the listed variables as integers, overriding their default type. If a variable name is an array, the INTEGER statement also specifies its size.

Examples

```
INTEGER ACCTNO,CHKNO
```

declares ACCTNO and CHKNO to be integer variables.

```
INTEGER ACCNO(1ØØ),CHKNO(2Ø,3,1Ø)
```

declares ACCNO and CHKNO to be integer arrays.

INTEGER*4
Declaring an extended integer variable.

Nonexecutable
Chapter 6

INTEGER*4 variable name(dimensions),
variable name(dimensions),...

Declares the listed variables to the extended integer variables. If the variable name is an array, you may include its dimensions in the statement.

Example

```
INTEGER*4  INVENT(1Ø),N
```

declares N to be an extended integer and INVENT to be an extended integer array consisting of 1Ø elements.

LOGICAL Nonexecutable
Declaring variables to be logical Chapter 6

LOGICAL variable(dimensions),variable(dimensions),..

Declares logical variable types (there are no default logical variables). If the variable represents an array, this statement specifies its dimension.

Example

```
LOGICAL L1,L2(5)
```

declares L1 to be logical and L2 to be a logical array.

OPEN
Opening a disk file

Executable
Chapter 8

CALL OPEN(logical unit number,'filename',
record length)

Calls a subroutine that opens a file for I/O. The logical unit number is an integer that corresponds to a legitimate disk LUN. The filename is any valid TRSDOS file name (note that it is enclosed is single quotes). The record length is an integer the value of which is between Ø and 256. The exact value of the record length is a function of the access mode (sequential or direct) and the type and amount of data.

Example

```
CALL OPEN (6,'TEST/DAT',25)
```

opens a file called TEST/DAT on a drive the records of which are to be 25 bytes long. The logical unit to be used as a buffer is LUN 6.

OUT
Direct access to the I/O ports

Executable
Chapter 8

```
CALL OUT(port, byte)
```

Serves to output the value of byte to the I/O port given by port.

Example

```
BYTE A
.
CALL OUT (21,A)
```

sends the value of A to I/O port 21.

PAUSE Executable
Pausing in the middle of a program

PAUSE string

Temporarily stops the program, at which time the message PAUSE or PAUSE string appears on the screen. The string can be any alphanumeric message up to six characters long. Pressing any key except <T> causes the program to continue. Pressing <T> terminates the program.

Example

```
PAUSE CHDSK
```

prints PAUSE CHDSK on the screen and execution of the program ceases until you press any key except <T>.

POKE Executable
"Poking" a value into memory

CALL POKE (integer,byte)

Inserts a value into memory. The integer specifies the memory location, and the byte represents the value to be POKEd.

Example

```
CALL POKE(16412,1)
```

POKEs the number 1 into memory location 16412 (hex 4Ø1C).

PROGRAM Nonexecutable
Naming a program

PROGRAM name

Titles your main program. This statement is optional, but if you use it, it must be the first statement in the program. The name must start with a letter and can include up to five more alphanumeric characters. If you do not use the PROGRAM statement, your computer automatically assigns the name $MAIN to the program.

Examples

```
PROGRAM TEST1Ø
```

titles the program TEST1Ø.

```
PROGRAM A1
```

titles the program A1.

Note: The name you choose for your program should not be the same name as a FORTRAN statement or function.

RAN Executable
Random number generator

RAN (real number)

Returns a random real number between Ø and 1. If the number in the argument is less than zero, the first value of a new sequence of random numbers is returned. If the number equals zero, then the last random number generated is returned. If the number is greater than zero, then the next number in the sequence is returned.

Examples

```
A = RAN(-1.Ø)
```

sets A equal to some new random number.

```
A = RAN(Ø.Ø)
```

sets A equal to the last random number generated.

```
A = RAN(1.Ø)
```

sets A equal to the next random number.

READ
Reading in a record

Executable
Chapter 8

READ(logical unit number,format statement label,
REC=record number,END=statement label,
ERR=statement label)variable list

Inputs data from the keyboard or from a diskette data file, depending on the logical unit number. The format statement label and its associated comma are optional. If you use them, the computer reads data in the format listed in that statement. If you do not use them, it reads the data in unformatted (only allowed for disk input).

The REC= option specifies the integer record number (for direct access files). The END= option gives the statement label of the next statement to be executed after reading the last record of the file. The ERR= option gives the statement label of the next statement to be executed in case of an input error (hardware error).

The variable list is optional. If you use it, the variables represent the symbolic name of the location in which the data will be stored. If you do not use it, the data is placed into the Holleriths of the FORMAT statement. You use this to change Hollerith strings in the FORMAT statement.

Example 1

```
READ(6,1Ø,REC=N,END=2Ø,ERR=1ØØ) A
```

In this example, 6 is the LUN, 1Ø is the FORMAT label, N is the number of the record to be read, 2Ø is the label of the statement to be executed after reading the last record, and 1ØØ is the label of the statement to be executed in case of an input error.

Example 2

You may use an "implied DO loop" as a variable list item. Its syntax is:

array name(subscript),integer variable=starting value, ending value,increment

It follows the same rules as a DO loop, but the implied DO loop lets an entire array be written on one record. For example:

```
          BYTE A(1Ø)
          READ(6,1Ø) (A(I),I=1,1Ø)
    1Ø    FORMAT(1ØA2)
```

The computer reads a string in from LUN 6 and places it into a byte array of 1Ø elements. This is equivalent to:

```
          READ(6,1Ø) A(1),A(2),A(3),A(4),A(5),
          A(6),A(7),A(8),A(9),A(1Ø)
```

(For more information on implied DO loops, see Chapter 8.)

Example 3

```
          READ(5,25)
    25    FORMAT (1ØHABCDEFGHIJ)
```

The next 1Ø characters input from the keyboard replace the 'ABCDEFGHIJ' in FORMAT statement 25.

Example 4

```
          READ(5,1Ø)A,B
    1Ø    FORMAT(F4.2,F4.1)
```

The computer stops and awaits input. You may either type the two values as one continuous line:

```
23.1b7.8
```

or separate them with commas:

```
23.1,7.8
```

Both set A equal to 23.1 and B equal to 7.8.

When using the continuous line, you must type some character (even if it's a blank space) for every character in the field.

The computer interprets the commas as the last character in the field.

REAL
Declaring a variable to be real

Nonexecutable
Chapter 6

REAL variable(dimension),variable(dimension),...

Declares the listed variables to be real. If the variable is an array name, the statement can also specify its size.

Examples

```
REAL ITEM,NUMBER
```

declares ITEM and NUMBER to be real variables.

```
REAL ITEM(1Ø),NUMBER(2Ø,2Ø)
```

declares ITEM and NUMBER to be real arrays.

RETURN
Returning from a subroutine or function

Executable
Chapter 9

RETURN

Represents the logical termination point of a subroutine or function. The subroutine or function can contain any number of RETURN statements but must have at least one.

Example

```
RETURN
```

REWIND Executable
Resetting the pointer in a file Chapter 8

REWIND logical unit number

Resets the pointer in a sequentially addressed file to the first record. The computer closes the file and then reopens it under the same specifications as before. The logical unit number (LUN) must reference an already opened, sequentially addressed data file.

Example

```
REWIND 6
```

closes the file associated with LUN 6 and then reopens it.

STOP Executable
Termination of a program

STOP string

Signifies the logical end of the program. The optional string after the STOP can be any six-character message. Your screen displays STOP string after the computer encounters the STOP. The STOP command is optional.

Example

```
STOP I/OERR
```

stops program execution and displays the message STOP I/OERR.

SUBROUTINE Nonexecutable
Defining a subroutine Chapter 9

SUBROUTINE subroutine name(local variable list)

Defines a subroutine subprogram unit and must be the first statement of that unit. Up to six alphanumeric characters constitute the subroutine name, the first of which must be a letter. The local variable list is optional. If you use them, the local variables must match in type and number the variable list in the CALL statement of the main program.

Example

```
SUBROUTINE SUB1(X,Y,Z)
```

defines a subroutine named SUB1 that uses the local variables X, Y, and Z.

WRITE
Outputting

Executable
Chapter 8

WRITE(logical unit number,format statement label,
REC=record number,ERR=statement label)
variable list

Outputs data to the printer, screen, or diskette, depending on the logical unit number. The format statement associated with WRITE specifies the type of output. It is optional. If you do not specify the type of output, the computer outputs unformatted data (only to the diskette).

The REC= option gives the integer record number of the location to write the data (for direct access files). The ERR= option specifies where to go in the program in case of an output error, such as a hardware error.

The variable list is optional. If you use it, the variables must match those in the corresponding FORMAT statement. If you do not use it, the computer prints or executes only what is in the FORMAT statement (literals, Holleriths, carriage control commands, or blank spaces).

Example 1

```
WRITE(6,1Ø,REC=1,ERR=2Ø) A,B
```

instructs the computer to write to record number 2 of a disk file (LUN 6) the variables A and B, using the format given in statement 1Ø. If an error occurs in the output, control transfers to Statement 2Ø.

Example 2

```
          DIMENSION A(1Ø)
          .
          DO 1Ø I=1,1Ø
          WRITE (7,REC=I)A(I)
    1Ø    CONTINUE
```

writes A(I) (unformatted) to record number I of a disk file (LUN 6).

Example 3

WRITE statements can use implied DO loops as variable lists. (See READ.)

```
          DIMENSION A(1Ø)
          .
          .
          WRITE (6,1Ø) (A(I),I=1,1Ø)
    1Ø    FORMAT (1ØF6.1)
```

writes the elements of the array A to LUN 6, using format 1Ø.

When using a implied DO loop in a WRITE statement, you cannot use a complex expression as the index. For example:

```
          WRITE(6,1Ø)(A(I-5),I=1,1Ø)
```

is invalid. (For more information on implied DO loops, see Chapter 8.)

Example 4

```
          WRITE (2,1Ø)
    1Ø    FORMAT ('1',2ØX,'TABLE OF VALUES')
```

This program gives no variable list, and only what is in format statement 1Ø, a carriage control command and a literal, is sent to the printer.

CHAPTER 11 / FORTRAN FUNCTIONS

A function is a built-in sequence of operations that FORTRAN performs on data. A function is actually a subroutine that returns data. The Compiler's functions save you from writing a routine, and they operate faster than a routine.

A function consists of its name and the data you specify. You must always enclose this data in parentheses, and, if more than one data item is required, you must separate the items with commas.

This chapter lists the syntax (format) to use in typing each function. If the data required is termed integer number, you may insert any integer expression that returns an integer value. If it is termed real number, you may insert any real expression. If is is termed double precision number, you may insert any double precision expression.

Important Note: Whenever you use a double precision number in a function, you must type it with the D exponential notation. (For more information, see Chapter 6.)

The value returned depends on the function used. Generally, all functions that are considered real variables return real values, those considered integer variables return integer values, and the functions beginning with the letter D return double precision values. For example:

ABS	returns a real value.
DABS	returns a double precision value.
IABS	returns an integer value.

Example

```
SQRT(A + 6.Ø)
```

computes the square root of A + 6.

Functions cannot stand alone in a FORTRAN program. You must use them in the same way you use data. For example:

```
A = SQRT(7.Ø)
```

sets A equal to the square root of 7.Ø.

FORTRAN contains the following library functions:

ABS	DCOS	IFIX
AINT	DEXP	INP
ALOG	DIM	INT
ALOG1Ø	DLOG	ISIGN
AMAXØ	DLOG1Ø	MAXØ
AMAX1	DMAX1	MAX1
AMINO	DMIN1	MINØ
AMIN1	DMOD	MIN1
AMOD	DSIGN	MOD
ATAN	DSIN	PEEK
ATAN2	DSQRT	SIGN
COS	EXP	SIN
DABS	FLOAT	SNGL
DATAN	IABS	SQRT
DATAN2	IDIM	TANH
DBLE	IDINT	

This chapter contains a listing of all these library functions, alphabetized by the "root" function name (the name underlined in bold face).

For all functions the degrees are expressed in radians.

Creating a Function

You can also create your own function in the following way by using this statement in your main program:

function name(local variable list) = expression

Example

```
ANSWER(A,B,C) = (A / B) - C
.
.
ANS = ANSWER(X,Y,Z)
```

ANSWER is the function name. A, B, and C are the local variables, and X, Y, and Z are the actual variables.

You must list this type of function in the main program before making any reference to it.

Another way to create a function is by using the FUNCTION statement to create a function subprogram. (See FUNCTION in Chapter 1Ø.)

ABS,DABS,IABS
Absolute value

IABS(integer number)
ABS(real number)
DABS(double precision number)

Return the absolute value of the number. IABS returns an integer, ABS returns a real number, and DABS returns a double precision number.

Examples

```
R = -39.4
A = ABS(R)
```

sets A equal to 39.4.

```
B = DABS(-942144.ØØ343DØ)
```

sets B equal to 942144.ØØ343.

```
J = IABS(-39)
```

sets J equal to 39.

ATAN,DATAN
Arctangent

ATAN(real number)
DATAN(double precision number)

Return the arctangent in radians of the number. ATAN returns a real value between -pi/2 and pi/2 radians, and DATAN returns double precision values between -pi/2 and pi/2 radians.

Examples

```
A = ATAN(1.Ø9)
```

sets A equal to Ø.8284338.

```
D = DATAN(1.Ø9DØ)
```

sets D equal to Ø.828433776420826.

ATAN2,DATAN2
Arctangent

ATAN2(real number-1,real number-2)
DATAN2(double precision-1,double precision-2)

Return the arctangent in radians of the first number divided by the second. ATAN2 returns real values between -pi/2 and pi/2 radians, and DATAN2 returns double precision values between -pi/2 and pi/2 radians.

Examples

```
A = ATAN2 (4.5,5.Ø)
```

sets A equal to Ø.7328151.

```
D = DATAN2 (4.5DØ,5.ØDØ)
```

sets D equal to Ø.732815ØØ17865Ø7.

COS,DCOS
Cosine

COS(real number)
DCOS(double precision number)

Return the cosine of the angle used in the argument. You must express the angle in radians, and it must be between Ø and (pi). COS returns a real value, and DCOS returns a double precision value.

Examples

```
A = COS(1.Ø)
```

sets A equal to Ø.54Ø3Ø23.

```
D = DCOS(1.ØØØØDØ)
```

sets D equal to Ø.54Ø3Ø23Ø5868140.

DBLE
Converting single precision to double precision

DBLE (real number)

Converts a real number to a double precision number by considering the insignificant digits of the real number significant.

Example

```
A = Ø.1322329
D = DBLE(A)
```

sets D equal to Ø.1322329ØØØØØØØØ.

DIM,IDIM
Positive difference

IDIM(integer-1,integer-2)
DIM(real-1,real-2)

Return the difference of the first number and the minimum of the two numbers. IDIM returns an integer. DIM returns a real number.

Examples

```
A = DIM(24.5,16.Ø)
```

sets A equal to 8.5.

```
A = DIM(16.Ø,24.5)
```

sets A equal to Ø.Ø.

```
I = IDIM(1Ø,2Ø)
```

sets I equal to Ø.

```
I = IDIM(35,-2)
```

sets I equal to 37.

EXP,DEXP
Raising "e" to a power

EXP(real number)
DEXP(double precision number)

Return the value of "e" (approximately 2.71828) raised to the power supplied by the argument. EXP returns a real value, and DEXP returns a double precisioin value.

Examples

```
A = 1.Ø
E = EXP(A)
```

sets E equal to 2.7183.

```
D = DEXP(1.ØØØØDØ)
```

sets D equal to 2.718281828459Ø5.

IFIX
Converts a real number to an integer

IFIX(real number)

Converts a real number to an integer (by truncation).
(This function is identical to INT.)

Example

```
J = 29.9
N = IFIX(J)
```

sets N equal to 29.

```
N = IFIX(-29.8)
```

sets N equal to -29.

FLOAT
Converts an integer to a real number

FLOAT(integer)

Converts an integer to a real number.

Example

```
A = FLOAT(2Ø)
```

sets A equal to 2Ø.ØØØØØ.

INP
Inputting data from port.

INP(port)

Returns the value from the port. The value is a byte.

Example

```
BYTE A
A = INP(1Ø)
```

sets A equal to the value in port 1Ø.

AINT,INT,IDINT
Truncation of a real or double precision number

IDINT(double precision number)
AINT(real number)
INT(real number)

Truncate the decimal portion of the number. INT and IDINT return integers, and AINT returns a real number. (INT is identical to IFIX.)

Note: For INT and IDINT, the number must be small enough in magnitude to fit in an integer type variable (-32768 to 32767).

Examples

```
A = AINT(34.33)
```

sets A equal to 34.ØØØØØ.

```
C = -19.63
I = INT(C)
```

sets I equal to -19.

```
J = IDINT(31994.6223DØ)
```

sets J equal to 31994.

ALOG,DLOG
Natural logarithm

ALOG(real number)
DLOG(double precision number)

Return the natural logarithm of the argument. ALOG returns a real value, and DLOG returns a double precision number.

Example

```
A = ALOG(45Ø99.93)
```

sets A equal to 1Ø.7166.

```
D = DLOG(45Ø99.93DØ)
```

sets D equal to 1Ø.7166359733831.

ALOG1Ø,DLOG1Ø
Common logarithm

ALOG1Ø(real number)
DLOG1Ø(double precision number)

Return the common (base 1Ø) logarithm of the number given in the function. (Because of the nature of logarithms, the number in the function must be positive.) ALOG1Ø returns a real number, and DLOG1Ø returns a double precision number.

Example

```
A = ALOG1Ø(45Ø99.93)
```

sets A equal to 4.654176.

```
D = DLOG1Ø(45Ø99.93DØ)
```

sets D equal to 4.65417586780618.

AMAXØ,AMAXØ,MAX1,DMAX1
Find the maximum in a list

MAXØ(integer list)
MAX1(real list)
AMAXØ(integer list)
AMAX1(real list)
DMAX1(double precision list)

Find the maximum value in a list of items. MAXØ finds the maximum value in an integer list and returns integer values. MAX1 returns integer values from a list of real numbers.

AMAXØ takes an integer list, finds the maximum value, and returns a real number. AMAX1 returns a real value from a list of real numbers.

DMAX1 returns the maximum double precision values from a list of double precision numbers.

Examples

```
A = AMAXØ(-34,59,1Ø)
```

sets A equal to 59.ØØØØØ.

```
A = AMAX1(99.4,2Ø.3,9.Ø)
```

sets A equal to 99.4ØØØØ.

```
N = MAXØ(25,-1Ø,3Ø)
```

sets N equal to 3Ø.

```
     N = MAX1(12.5,1Ø.Ø,131.3,1.3)
```

sets N equal to 131.

```
     D = DMAX1(13.2232DØ,1ØØ.3211DØ,-1.ØØ2311DØ)
```

sets D equal to 1ØØ.3211ØØØØØØØØ.

AMINØ,AMIN1,DMIN1,MINØ,MIN1
Find the minimum in a list

MINØ(integer list)
MIN1(real list)
AMINØ(integer list)
AMIN1(real list)
DMIN1(double precision list)

Return the minimum value of a list of numbers. MINØ and MIN1 return integer values, AMINØ and AMIN1 return real values, and DMIN1 returns double precision values.

Examples

```
A = AMINØ(-34,59,1Ø)
```

sets A equal to -34.ØØØØØ.

```
A = AMIN1(99.4,2Ø.3,9.Ø)
```

sets A equal to 9.ØØØØØ.

```
N = MINØ(25,-1Ø,3Ø)
```

sets N equal to -1Ø.

```
N = MIN1(12.5,1Ø.Ø,131.3,-1.3)
```

sets N equal to -1.

```
D = DMIN1(13.2232DØ,1ØØ.3211DØ,-1.ØØ2311DØ)
```

sets D equal to -1.ØØ2311ØØØØØØØØ,

AMOD,DMOD,MOD
Arithmetic remainder

MOD(integer dividend,integer divisor)
AMOD(real dividend,real divisor)
DMOD(double precision dividend,double precision divisor)

Return the remainder left after the division of the dividend by the divisor. MOD returns an integer, AMOD returns a real number, and DMOD returns a double precision number.

Examples

```
     RMAIN = AMOD(13.6,2.Ø)
```

sets RMAIN equal to 1.6ØØØØØ.

```
     IMAIN = MOD(25,2)
```

sets IMAIN equal to 1.

```
     DMAIN = DMOD(1Ø24.ØØ39DØ,2.ØØØDØ)
```

sets DMAIN equal to Ø.ØØ3899999894ØØ2.

PEEK
"Peeking" at a memory location

PEEK(integer)

Returns the value stored at the memory location given by the integer. The value is a number in the range -127 to 128.

Example

```
A = PEEK(32ØØØ)
```

sets A equal to the value stored at memory location 32ØØØ (hexadecimal 7DØØ).

DSIGN,ISIGN,SIGN
Transfer signs

ISIGN(integer-1,integer-2)
SIGN(real-1,real-2)
DSIGN(double precision-1,double precision-2)

Transfer the sign of the second number to the first number (that is, the magnitude of the first times the sign of the second). ISIGN returns an integer, SIGN returns a real number, and DSIGN returns a double precision number.

Examples

```
N = ISIGN(-292,14)
```

sets N equal to 292.

```
A = SIGN(-1Ø.3,-59.2)
```

sets A equal to -1Ø.3ØØØ.

```
D = DSIGN(Ø.14423332D4,-Ø.12133DØ)
```

sets D equal to -.1442333ØØØØØØØ.

DSIN,SIN
Sine of an angle

SIN(real number)
DSIN(double precision number)

Return the sine of the angle given in the argument. You must give the angle in radians, and it must be between Ø and pi. SIN returns a real number, and DSIN returns a double precision number.

Examples

```
B = Ø.55
A = SIN(.55)
```

sets A equal to Ø.5226871.

```
D = DSIN (.55ØØØDØ)
```

sets D equal to Ø.52268722Ø93Ø659.

SNGL
Converting double precision to single precision

SNGL(double precision number)

Converts a double precision number to a real number by saving the most significant digits.

Example

```
A = SNGL(.143294Ø48742Ø4DØ)
```

sets A equal to .1432941, but only the first seven digits are considered significant.

DSQRT,SQRT
Square root

SQRT(real number)
DSQRT(double precision number)

Return the square root of the number. (The number must be positive.) SQRT returns a real number value, and DSQRT returns a double precision number.

Examples

```
A = SQRT(81.Ø)
```

sets A equal to 9.ØØØØØØ.

```
DNUM = (25.ØØØ23113DØ)
D = DSQRT(DNUM)
```

sets D equal to 5.ØØØØ231129466.

TANH
Hyperbolic Tangent

TANH(real number)

Returns the real number value of the hyperbolic tangent of the angle given in the argument. You must express the angle in radians, and it must be between -(pi)/2 and (pi)/2.

Example

```
A = TANH(1.Ø)
```

sets A equal to Ø.7616.

Section III

Error Messages

Part 3

ERROR MESSAGES

A/ TRSDOS ERRORS

Important Note: Since your operating system is TRSDOS, some of the errors you encounter while running the FORTRAN package may be TRSDOS errors. TRSDOS 6 displays the message and not the number of the error.

TRSDOS Error Codes/Messages

Decimal	Hex	Message
Ø	X'ØØ'	No error
1	X'Ø1'	Parity error during header read
2	X'Ø2'	Seek error during read
3	X'Ø3'	Lost data during read
4	X'Ø4'	Parity error during read
5	X'Ø5'	Data record not found during read
6	X'Ø6'	Attempted to read system data record
7	X'Ø7'	Attempted to read locked/deleted data record
8	X'Ø8'	Device not available
9	X'Ø9'	Parity error during header write
1Ø	X'ØA'	Seek error during write
11	X'ØB'	Lost data during write
12	X'ØC'	Parity error during write
13	X'ØD'	Data record not found during write
14	X'ØE'	Write fault on disk drive
15	X'ØF'	Write protected disk
16	X'1Ø'	Illegal logical file number
17	X'11'	Directory read error
18	X'12'	Directory write error
19	X'13'	Illegal file name
2Ø	X'14'	GAT read error
21	X'15'	GAT write error
22	X'16'	HIT read error
23	X'17'	HIT write error
24	X'18'	File not in directory
25	X'19'	File access denied
26	X'1A'	Full or write protected disk
27	X'1B'	Disk space full
28	X'1C'	End of file encountered

29	X'1D'	Record number out of range
3Ø	X'1E'	Directory full - can't extend file
31	X'1F'	Program not found
32	X'2Ø'	Illegal drive number
33	X'21'	No device space available
34	X'22'	Load file format error
37	X'25'	Illegal access attempted to protected file
38	X'26'	File not open
39	X'27'	Device in use
4Ø	X'28'	Protected system device
41	X'29'	File already open
42	X'2A'	LRL open fault
43	X'2B'	SVC parameter error
63	X'3F'	Extended error
--		Unknown error code

B/ COMPILER (F8Ø) ERROR MESSAGES

The two kinds of Compiler error messages are runtime and Compiler. Compiler errors occur when the program is being compiled; runtime errors occur when you execute the program.

Compiler Errors

Your FORTRAN Compiler detects two kinds of errors: Warnings and Fatal Errors.

When you receive a Warning Error, compilation continues with the next item on the source line. (Remember, your computer cannot directly process source lines. You must compile them into an object program consisting of instructions in a machine language that your computer can understand.)

Example

```
%LINE 16:  Missing Integer Variable
```

When a Fatal Error occurs, the Compiler ignores the rest of the statement line, including any of the statement line's continuation lines.

Examples

```
?LINE:  3 Statement Is Out Of Sequence:  Format'
        ?1 Fatal Error(s) Detected
(or)
?LINE:  3 Premature End Of File On Input Device:
        ?1 Fatal Error(s) Detected
```

Percent signs (%) precede Warning Errors; question marks (?) precede Fatal Errors. Your Compiler displays the physical line number next and then the error code or error message.

FATAL ERRORS

Note: Words capitalized in the following error messages are reserved words in FORTRAN and have special significance to your Compiler.

Fatal Compiler Errors (in alphabetical order)

MESSAGES

Backwards DO reference
Consecutive Operators
Data Pool Overflow
Function Call wilth No Parameters
Identifier Too Long
Illegal Character for Syntax
Illegal Data Constant
Illegal DO Nesting
Illegal Hollerith Constsruction
Illegal Integer Quanity
Illegal Item Following INTEGER or REAL or LOGICAL
Illegal Logical Form Operator
Illegal Mixed Mode Operation
Illegal Operator
Illegal Procedure Name
Illegal Statement Completion
Illegal Statement Following Logical IF
Illegal Statement Function Name
Illegal Statement Number
Improper Subscript Syntax
Incorrect Integer Constant
Incorrect Number of DATA Constants
Invalid DATA Constant or Repeat Factor
Invalid Data List Element in I/O
Invalid Logical Operator
Invalid Statement Number
Literal String Too Large
Mismatched Parentheses
Missing Integer Quantity
Missing Name
Not a Variable Name

Premature End of File on Input Device
Stack Overflow
Statement Out of Sequence
Statement Unrecognizable or Misspelled
Unbalanced DO Nest

Compiler Nonfatal Errors

Array Multiple EQUIVALENCEd within a Group
Array Name Expected
Array Name Misuse
Block Name = Procedure Name
Code Output in BLOCK DATA
COMMON Base Lowered
COMMON Name Usage
Division by Zero
Duplicte Statement Label
Empty List for Unformatted WRITE
Format Nest Too Deep
Function with no Parameter
Hex Constant Overflow
Illegal Argument to ENCODE/DECODE
Illegal DO Termination
Invalid Operand Usage
Invalid Statement Number Usage
Missing DO Termination
Missing Integer Variable
Missing Statement Number on FORMAT
Mixing of Operand Modes Not Allowed
Multiple EQUIVALENCE of COMMON
No Path to this Statement
Non-COMMON Variable in BLOCK DATA
Non-Integer Expression
Operand Mode Not Compatible with Operation
RETURN in a Main Program
Statement Number Not FORMAT Associated
STATUS Error on READ
Undefined Labels Have Occurred
Wrong Number of Subscripts
Zero Format Factor
Zero Repeat Factor

Compiler Runtime Error Messages

When you execute your FORTRAN program, you may receive Runtime Warning Errors and Runtime Fatal Errors. After a maximum of 2Ø Runtime Warning Errors, execution ceases; before this, execution continues after the warning. A Runtime Fatal Error causes execution to cease. Runtime errors are in two-character code and enclosed with asterisks:

TL

RUNTIME WARNING ERRORS

A2	Both Arguments of ATAN2 are Ø
BE	Binary Exponent Overflow
BI	Buffer Size Exceeded During Binary I/O
CN	Conversion Overflow
DE	Decimal Exponent Overflow (Number in input stream had an exponent larger than 99)
IB	Input Buffer Limit Exceeded
IN	Input Record Too Long
IO	Illegal I/O Operation
IS	Integer Size Too Long
OB	Output Buffer Limit Exceeded
OV	Arithmetic Overflow
RC	Negative Repeat Count in FORMAT
SN	Argument to SIN Too Large
TL	Too Many Left Parentheses in FORMAT

RUNTIME FATAL ERRORS

DT	Date Type Does Not Agree With FORMAT Specification
DZ	Division by Zero, REAL or INTEGER
EF	EOF Encountered on READ
FO	FORMAT Field Width is Zero
FW	FORMAT Field Width is Too Small
ID	Illegal FORMAT Descriptor
IT	I/O Transmission Error

LG	Illegal Argument to LOG Function (Negative or Zero
ML	Missing Left Parenthesis in FORMAT
MP	Missing Period in FORMAT
SQ	Illegal Argument to SQRT Function (Negative)

C/ EDITOR (ALEDIT) ERROR MESSAGES

BAD FILE FORMAT

The file is not a type ALEDIT can load, either fixed LRL 1 or Variable, and with record length not greater than 256 bytes.

BAD FILENAME FORMAT

The filename is too long or incorrectly formatted on a load or a write command.

BAD PARAMETERS

The ASCII line number converted to hexadecimal is greater than 65535 decimal (for line number request).

The change string is zero or the length of the line to be changed is zero (for Change command).

BUFFER FULL

The edit buffer has no more room. Program returns from any mode to Command Mode. **Note:** The edit buffer is about 4K smaller if DO, HOST, COMM, SPOOL, DEBUG, or ALBUG are on.

LINE LENGTH TOO LONG, TRUNCATING LINE

You are loading a file that has lines longer than 78 characters.

LINE NUMBER TOO LARGE

The line number is larger than the last line number in the file. The editor does not recognize your command. Retype it.

NO TEXT

The edit buffer is empty. The only commands that are effective are: K, L, Y, I, Q, J, S.

OCCURRENCE TOO LARGE

In the Find and Change commands the occurrence is greater than 255.

SEARCH ARG TOO LONG

The string you want to search for is longer than 37 characters.

SYNTAX ERROR

The command is improperly specified.

TOTAL LINE LENGTH TOO LONG

The new line created by a Change command is greater than the acceptable Line Length.

If the Editor returns an error code, it is a TRSDOS error message. You can identify it by simply typing in the error number. For example, at TRSDOS Ready, type:

ERROR 19 <ENTER>

or at the Editor Command Mode, type:

S ERROR 19 <ENTER>

and your computer answers you with the correct identification:

INVALID FILE NAME

You can do this any time your computer identifies an error of which you are not aware.

HIT ANY KEY TO CONTINUE

If there is an error in the load or write routines, the Editor waits for the user to read the entire error message.

D/ LINKER (L8Ø) ERROR MESSAGES

Your Linker has the following error messages:

Origin Above Loader Memory, Move Anyway (Y or N)?
Origin Below Loader Memory, Move Anwyay (Y or N)?

When you enter a -E or -G switch and you receive this error, either the data or program area has an origin or top that lies outside loader memory (that is, loader origin to top of memory). If you type Y <ENTER>, the Linker moves the area and continues. If you enter anything else, the Linker exits. In either case, if you enter a -N, the image will already have been saved.

%Mult. Def. Global YYYYYY

This error occurs when more than one definition for the global (internal) symbol YYYYYY is encountered during the loading process. Attempting to link two main programs may cause this error. (Use the R switch to reset the Linker before linking another main program.) Using a variable name that is the same as a global symbol name also causes this error.

Note: If you link a program that does not have either a READ or WRITE statement, the Linker gives you "undefined global(s)" errors. Ignore this message. It has no effect on the execution of your program.

%Overlaying Program Area ,Start = xxxx
Data ,Public = <symbol name>(xxxx)
,External = <symbol name>(xxxx)

A -D or -P switch destroys data you have already loaded.

%2nd COMMON Larger /XXXXXX/

This error occurs if the first definition of COMMON block /XXXXXX/ is not the largest definition. Reorder module loading sequence or change COMMON block definitions.

?Can't Save Object File

A disk error occurs when you are saving the file.

?Command Error

This error occurs when a Linker command is unrecognizable.

?<file> Not Found

This error occurs when a filename you give in a command string does not exist.

?Intersecting Program Area
Data

This error occurs when your program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

?Loading Error

This error occurs when the last file given for input is not a properly formatted Linker object file.

?Out of Memory

This error occurs if there is not enough memory to load the program.

?Start Symbol - <name> - Undefined

This error occurs if after you enter a -E: switch the symbol specified is not defined.

Section IV

Quick Reference

Part 4

QUICK REFERENCE

A/EDITOR

Description of Terms

current line

line where the cursor is currently positioned.

del (delimiter)

one of the following characters which marks the beginning and ending of a string:

! " # $ % & ' ( ) * + , - . / : ; < = > ?

string

one to thirty-seven ASCII characters.

text

source program or text currently in RAM.

Command Mode

Command	Description
#line	Moves the cursor to specific line number and moves that line to the top of the screen.
. (period)	Displays the current line sequence number.
<up arrow>	Moves the cursor up one line.

<down arrow>	Moves the cursor down one line.
<left arrow>	Moves the cursor one character to the left.
<right arrow>	Moves the cursor one character to the right.
1	Marks the begining of a block.
2	Marks the last line of a block.
3	Cancels block markers.
A <ENTER>	Reexecutes the last (C, F, X, L, W) executed command.
B	Moves the cursor to the bottom of the text.
<BREAK>	Cancels any command and returns to Command Mode.
C del string1 del string2 del occurrence <ENTER>	Changes string1 to string 2 for a specified number of occurrences.
<CTRL> A	Moves the cursor to the top of the screen.
<CTRL> B	Moves the cursor to the bottom of the screen (or to the line after the last line of text).
D	Deletes the current line or marked block of lines (you need not press <ENTER>).
E	Enters Line Edit Mode.
F del string del occurrence <ENTER>	Finds the specified occurrence of string. If you omit occurrence, it finds the first occurrence.
G <ENTER>	Deletes all text from the current line to the end.

H <ENTER>	Prints the entire text if entered as the first command or the specified block on the printer.
I	Enters Insert Mode.
J	Displays the current size of text and remaining memory.
K	Deletes all text.
L filespec $C	Loads a new file into the Editor. $C is optional.
M	Moves a marked block ahead of the current line.
N	Updates the display.
O	Copies a block of text.
P	Moves the cursor to the next page (a page is 24 lines).
Q <ENTER>	Quits (exits) the Editor
R <ENTER>	Deletes the current line and enters Insert Mode.
<SHIFT> <up arrow>	Cancels the current command line if you have not yet pressed <ENTER>.
T	Moves the cursor to the top of text.
U	Moves the cursor to the previous page (a page is 24 lines).
V	Scrolls the current line to the top of the screen.

W filespec $option. . . <ENTER>
Saves a program on disk. $E exits the Editor after saving the file. $M saves the file as a fixed length record file with an LRL of 256.

X del string1 del string2 del occurrence
Changes string1 to string 2 for the number of occurrences you specify but prompts before making the change.

Line Edit Mode

Command	Description
<left arrow>	Moves cursor one position to the left.
<right arrow>	Moves cursor to the next tab position while in I, X, or H subcommands.
A	Clears all changes and reenters Edit Mode for the current line.
E	Exits Edit Mode and stores changes.
<ENTER>	Identical to the E subcommand.
H string	Deletes remaining characters, enters Insert Mode, and lets you insert a string.
I string	Lets you insert at the current position of the cursor on the line. <left arrow> deletes characters from the line.
L	Moves the cursor to the beginning of the line.

n C string	Changes next n characters in the specified string. If you omit n, only one character is changed. <SHIFT> <up arrow> exits change early.
n D	Deletes n characters. If you omit n, only one character is deleted.
n K character	Kills all characters preceding the nth occurrence of the character. If you omit n, the first occurrence is used. If no match is found, the remainder of the line is killed.
n S character	Positions the cursor at the nth occurrence of character.
Q	Quits Edit Mode and cancels all changes.
<SHIFT> <up arrow>	Returns to Edit Command Mode from the I, X, C, or H subcommands.
<SPACEBAR>	Moves the cursor one position to the right.
X string	Moves the cursor to the end of the line, enters Insert Mode, and lets you insert a string.

Insert Mode

Command	Description
I	Places the Editor in Insert Mode and lets you enter program text.
<BREAK>	Exits Insert Mode and returns you to the ALEDIT Command Mode.

B/ COMPILER

Your commands tell the Compiler the name of the source file you want to compile and what options you want to use. Here is the format for a Compiler command:

Object filename, listing, filename=source filename-switch

- object filename -- This is the name you give your object file. It is optional. To create a relocatable object file, you must include this part of the command. The default extension for the object filename is /REL.
- listing filename -- This is the name you give the listing file. It is optional. The default extension for the listing file is /LST.
- source filename -- This is the name of a FORTRAN program you saved on diskette. The default extension for a FORTRAN source filename is /FOR. In a Compiler command you must always precede the source filename with an equal sign.
- switches -- This is the way you want the file command. It is optional.

Switches

Switch	Action
-O	Prints all listing addresses in octal.
-H	Prints all listing addresses in hexadecimal (default).
-N	Does not list the object code that is generated in the listing file. Lists only the FORTRAN source code.
-P	Each -P allocates an extra 1ØØ bytes of stack space for use during compilation. Use -P if stack overflow errors occur during the compilation. Otherwise, you do not need this switch.

-M	Specifies to the Compiler that the generated code should be in a form that can be loaded into ROMs.

Example

```
SAMPLE,SAMPLE=SAMPLE-P
```

C/ LINKER

Syntax

command filename-N, object filename-switches
command filename-N -- creates a TRSDOS command file. This is optional. If you use it, you must also use the -E. The default extension is /CMD.
object filename -- This is one or more relocatable object file that needs to be input. If you omit it, the Linker uses the previously loaded object file. The default extension is /REL.
switches -- Specify what action the Linker takes with the object file. If you omit switches, the Linker uses the -U switch.

Switches

The Linker switches are:

-R resets
-P or -D specify program and data area
-N saves command file
-U lists origin and end of program; undefined globals
-M lists origin and end; defined and undefined globals
-E exits the Linker to TRSDOS
-S searches line for globals.

Example

EXAMPLE-N,EXAMPLE-E

D/ FORTRAN STATEMENTS AND FUNCTIONS

ABS(real number)
Returns absolute value of a real number.

AINT(real number)
Truncates a real number.

ALOG(real number)
Returns the natural logarithm of the argument.

ALOG10(real number)
Returns the common logarithm of the real number.

AMAX0(integer list)
Finds the maximum in a list.

AMAX1(real list)
Finds the maximum in a list.

AMIN0(integer list)
Finds the minimum in a list.

AMIN1(real list)
Finds the minimum in a list.

AMOD(real dividend,real divisor)
Returns the arithmetic remainder.

ASSIGNinteger constant TO integer variable
Sets values for ASSIGNed GO TOs.

ATAN(real number)
Returns the arctangent in radians of a real number.

ATAN2(real number-1,real number -2)
Returns the arctangent in radians of the quantity of the first number divided by the second.

BLOCK DATA subprogram name
Titles BLOCK DATA subprograms.

BYTE variable(dimension) list
Declares a byte variable.

CALL subroutine name (variable list)
Accesses a subroutine.

CALL OPEN(logical unit number,'filename',
record length)
Opens a disk file.

CALL OUT(logical unit number,byte)
Directs output to the I/O ports.

CALL POKE(integer,byte)
"Pokes" a value into memory.

COMMON variable list
or
COMMON /block area name/variable list . . .
Declares COMMON location.

CONTINUE
A "No-operation" executable statement.

COS(real number)
Returns the cosine of the angle in radians used in the argument.

DABS(double precision number)
Returns the absolute value for a double precision number.

DATA variable list/data list/variable list/data list/...
Initializes variables.

DATAN(double precision number)
Returns the arctangent in radians of a double precision number in radians.

DATAN2(double precision-1,double precision-2)
Returns the arctangent in radians (double precision) of the quantity of the first number divided by the second in radians.

DBLE(real number)
Converts a single precision number to double precision.

DCOS(double precision number)
Returns the cosine of the angle in radians used in the argument.

DECODE(array name,format statement label)
variable list
Changes internal format.

DEXP(double precision number)
Raises "e" to a power.

DIM(real-1,real-2)
Returns the positive difference.

DIMENSION array name (dimension),array name(dimensions)
Dimensions a variable.

DLOG(double precision number)
Returns the natural logarithm of the argument.

DLOGIØ(double precision number)
Returns the common logarithm of a double precision number.

DMAX1(double precision list)
Finds the maximum in a list.

DMIN1(double precision list)
Finds the minimum in a list.

DMOD(double precision dividend,double precision divisor)
Returns an arithmetic remainder.

DO statement label integer variable=starting value,
ending value,increment
Looping.

DOUBLE PRECISION variable(dimensions),
variable(dimensions). . .
Declares a variable to be double precision.

DSIGN(double precision-1,double precision-2)
Transfers sign.

DSIN(double precision number)
Returns the sine of an angle.

DSQRT(double precision number)
Returns a square root.

ENCODE(array name,format statement label)variable list
Changes internal format.

END
Terminates the program.

ENDFILE logical unit number
Closes a file.

EQUIVALENCE (variable list), (variable list),...
Declares equivalent memory locations.

EXP (real number)
Raises "e" to a power.

EXTERNAL subprogram name list
Uses functions inside functions.

FLOAT(integer)
Converts an integer to a real number.

label FORMAT(specificaion list)
Formats input/output to the specification list.

FUNCTION function name(variable list)
Defines a function.

GO TO statement label
Unconditional GO TOs.

GO TO (statement label list), integer variable
Computed GO TOs.

GO TO integer variable,(statement label list)
Assigned GO TOs.

IABS(integer number)
Returns the absolute value for an integer number.

IDIM(integer-1,integer-2)
Returns the positive difference (integer).

IDINT(double precision number)
Truncates a double precision number.

IF (expression) label-1,label-2,label-3
Arithmetic IF.

IF (logical expression) executable statement
Logical IF.

IFIX(real number)
Converts a real number to an integer.

IMPLICIT type(range),type(range)...
Declares a range of default variable types.

INCLUDE filename
Brings in outside programs.

INP(logical unit number)
Inputs a value from the I/O ports.

INT(real number)
Truncates a real number.

INTEGER variable(dimensions),variable(dimensions),...
Declares a variable to be an integer.

INTEGER*4 variable(dimensions),variable(dimensions),...
Declares a variable to be an extended integer.

ISIGN(integer-1,integer-2)
Transfers signs.

LOGICAL variable(dimensions),variable(dimensions),...
Declares variables to be logical.

MAXØ(integer list)
Finds the maximum in an integer list.

MAX1(real list)
Finds the maximum in a real number list.

MINØ(integer list)
Finds the minimum in an integer list.

MIN1(real list)
Finds the minimum in a real number list.

MOD(integer dividend,integer divisor)
Returns the arithmetic remainder.

OPEN
See CALL OPEN.

OUT
See CALL OUT.

PAUSE string
Pauses in the middle of a program.

PEEK(integer)
"Peeks" at a memory location.

POKE
See CALL POKE.

PROGRAM name
Names a program.

RAN(real number)
Returns a random real number.

READ(logical unit number,format statement label,
REC=record number,END=statement label,
ERR=statement label)variable list
Reads in a record.

REAL variable(dimensions),variable(dimension),...
Declares a variable to be real.

RETURN
Returns from a subroutine or function.

REWIND logical unit number
Resets the pointer in a file.

SIGN(real-1,real-2)
Transfers signs.

SIN(real number)
Returns the sine of an angle.

SNGL(double precision number)
Converts double precision to single precision.

SQRT(double precision number)
Returns a square root.

STOP string
Terminates a program.

SUBROUTINE subroutine name(local variable list)
Defines a subroutine.

TANH(real number)
Returns the tangent.

WRITE(logical unit number,format statement label,
REC=record number,ERR=statement label)variable list
Outputs data to printer, screen, or diskette.

Section V

Appendices

APPENDICES

A/ Language Extensions and Restrictions

The FORTRAN language includes the following extensions to ANSI Standard FORTRAN (X3.9-1966).

1. If you use c in a 'STOP c' or 'PAUSE c' statement, c may be any six ASCII characters.
2. You may specify Error and End-of-File branches in READ and WRITE statements using the ERR= and END= options (although END is not used in WRITE).
3. The standard subprograms PEEK, POKE, INP, and OUT have been added to the FORTRAN library.
4. Statement functions may used subscripted variables.
5. You may use hexadecimal constants wherever Integer constants are normally allowed.
6. You may use the literal form of Hollerith data (character string between apostrophe characters -- single quotes) in place of the standard nH form.
7. You may use Holleriths and Literals in expressions in place of Integer constants.
8. The number of continuation lines is not restricted.
9. You may use mixed mode expressions and assignments. Conversions are done automatically.

FORTRAN-8Ø places the following restrictions on Standard FORTRAN.

1. The COMPLEX data type is not implemented. It may be included in a future release.
2. The specification statements must appear in the following order:

   A. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA
   B. Type, EXTERNAL, DIMENSION
   C. COMMON
   D. EQUIVALENCE
   E. DATA
   F. Statement Functions

3. A different amount of computer memory is allocated for each data type: Integer, Real, Double Precision, Logical.

4. The equal sign of a replacement statement and the first comma of a DO statement must appear on the initial statement line.

Descriptions of these language extensions and restrictions are included at the appropriate points in the text of this manual.

B/ I/O Interface

Input/Output operations are table-dispatched to the driver routine for the proper Logical Unit Number.

$LUNTB is the dispatch table. It contains one two-byte driver address for each possible LUN. It also has a one-byte entry at the beginning, which contains the maximum LUN plus one.

The initial runtime package provides for 1Ø LUN's (1-2Ø), all of which correspond to the TTY. You may redefine any of these or add more by changing the appropriate entries in $LUNTB and adding more drivers. The runtime system uses LUN3 for errors and other user communication. Therefore, LUN3 should correspond to the operator console. The initial structure of $LUNTB is shown in the listings following this appendix.

The device drivers also contain local disptach tables. Note that $LUNTB contains one address for each device, yet there are really seven possible operations per device:

(1) Formatted Read
(2) Formatted Write
(3) Binary Read
(4) Binary Write
(5) Rewind
(6) Backspace
(7) Endfile

Each device driver contains up to seven routines. The starting address of each routine is placed at the beginning of the driver, in the exact order listed above. The entry in $LUNTB is then pointed to this local table, and the runtime system indexes into it to get the address of the appropriate routine to handle the requested I/O operation.

The following conventions apply to the individual I/O routines:

1. Location $BF contains the data buffer address for READs and WRITEs.
2. For a WRITE, the number of bytes to write is in location $BL.
3. For a READ, the number of bytes should be returned in $BL.
4. All I/O operations set the condition codes before exit to indicate an error condition, end-of-file condition, or normal return:

   a) CY=1, Z=don't care -- I/O error
   b) CY=Ø, Z=Ø -- end-of-file encountered
   c) CY=Ø,Z=1 -- normal return

   The runtime system checks the condition codes after calling the driver. If they indicate an abnormal condition, control passes to the label specified by "ERR=" or "END=" or, if no label is specified, a Fatal Error results.
5. $IOERR is a global routine that prints an "ILLEGAL I/O OPERATION" message (nonfatal). You may use this routine if some operations are not allowed on a particular device (for example, Binary I/O on a TTY).

```
      MAC80 1.0       PAGE    1

                      00100  ;       TTY I/O DRIVER
                      00200
0000                  00300          EXT     $IOERR,$BL,$BF,$ERR,$TTYIN,$TTYOT
0012                  00400  IRECER  EQU     022     ;INPUT RECORD TOO LONG
0000                  00500          ENTRY   $DRV3
0000  0013 '          00600  $DRV3:  DW      DRV3FR  ;FORMATTED READ
0002  0042 '          00700          DW      DRV3FW  ;FORMATTED WRITE
0004  0010 '          00800          DW      DRV3BR  ;BINARY READ
0006  0010 '          00900          DW      DRV3BW  ;BINARY WRITE
0008  000E '          01000          DW      DRV3RE  ;REWIND
000A  000E '          01100          DW      DRV3BA  ;BACKSPACE
000C  000E '          01200          DW      DRV3EN  ;ENDFILE
000E  AF              01300  DRV3EN: XRA     A       ;THESE OPERATIONS ARE
                      01400                          ;NO-OPS FOR TTY
000E                  01500  DRV3RE  EQU     DRV3EN
000E                  01600  DRV3BA  EQU     DRV3EN
000F  C9              01700          RET
0010  C3 0000 *       01800  DRV3BW: JMP     $IOERR  ;ILLEGAL OPERATIONS
                      01900                          ;(PRINT ERROR AND RETURN)
0010                  02000  DRV3BR  EQU     DRV3BW
0013  AF              02100  DRV3FR: XRA     A       ;READ
0014  32 0000 *       02200          STA     $BL     ;ZERO BUFFER LENGTH
0017  CD 0000 *       02300  DRV31:  CALL    $TTYIN  ;INPUT A CHAR
001A  E6 7F           02400          ANI     0177    ;AND OFF PARITY
001C  FE 0A           02500          CPI     10      ;IGNORE LINE FEEDS
001E  CA 0017 '       02600          JZ      DRV31
0021  F5              02700          PUSH    PSW     ;SAVE IT
0022  2A 0015 *       02800          LHLD    $BL     ;GET CHAR POSIT IN BUFFER
0025  26 00           02900          MVI     H,0     ;ONLY 1 BYTE
0027  EB              03000          XCHG
0028  2A 0000 *       03100          LHLD    $BF     ;GET BUFFER ADDR
002B  19              03200          DAD     D       ;ADD OFFSET
002C  F1              03300          POP     PSW     ;GET CHAR
002D  77              03400          MOV     M,A     ;PUT IT IN BUFFER
002E  13              03500          INX     D       ;INCREMENT $BL
002F  EB              03600          XCHG
0030  22 0023 *       03700          SHLD    $BL     ;SAVE IT
0033  FE 0D           03800          CPI     015     ;CR?
0035  C8              03900          RZ              ;YES--DONE
0036  7D              04000          MOV     A,L     ;$BL
0037  FE 80           04100          CPI     128     ;MAX IS DECIMAL 128
0039  DA 0017 '       04200          JC      DRV31   ;GET NEXT CHAR
003C  CD 0000 *       04300          CALL    $ERR
003F  12              04400          DB      IRECER  ;INPUT RECORD TOO LONG
0040  AF              04500          XRA     A       ;CLEAR FLAGS
0041  C9              04600          RET
0042  3A 0031 *       04700  DRV3FW: LDA     $BL     ;BUFFER LENGTH
0045  B7              04800          ORA     A

      MAC80 1.0       PAGE    2

0046  C8              04900          RZ              ;EMPTY BUFFER
0047  2A 0029 *       05000          LHLD    $BF     ;BUFFER ADDRESS
004A  3D              05100          DCR     A       ;DECREMENT LENGTH
004B  F5              05200          PUSH    PSW     ;SAVE IT
004C  3E 0D           05300          MVI     A,13    ;CR
004E  CD 0000 *       05400          CALL    $TTYOT  ;OUTPUT IT
0051  7E              05500          MOV     A,M     ;GET FIRST CHAR IN BUFFER
0052  FE 2B           05600          CPI     '+'
```

```
0054   CA 0079 '      05700            JZ      DR3FW2   ;NO LINE FEEDS
0057   FE 31          05800            CPI     '1'
0059   C2 0064 '      05900            JNZ     DR3FW1   ;NOT FORM FEED
005C   3E 0C          06000            MVI     A,12     ;FORM FEED
005E   CD 004F *      06100            CALL    $TTYOT   ;OUTPUT IT
0061   C3 0079 '      06200            JMP     DR3FW2
0064   3E 0A          06300   DR3FW1:  MVI     A,10     ;LF
0066   CD 005F *      06400            CALL    $TTYOT
0069   7E             06500            MOV     A,M      ;GET CHAR BACK
006A   FE 20          06600            CPI     ' '
006C   CA 0079 '      06700            JZ      DR3FW2   ;NO MORE LINE FEEDS
006F   FE 30          06800            CPI     '0'
0071   C2 0079 '      06900            JNZ     DR3FW2   ;NO MORE LINE FEEDS
0074   3E 0A          07000            MVI     A,10     ;LF
0076   CD 0067 *      07100            CALL    $TTYOT
0079   F1             07200   DR3FW2:  POP     PSW      ;GET LENGTH BACK
007A   23             07300            INX     H        ;INCREMENT PTR
007B   C8             07400   DRV32:   RZ
007C   F5             07500            PUSH    PSW      ;SAVE CHAR COUNT
007D   7E             07600            MOV     A,M      ;GET NEXT CHAR
007E   23             07700            INX     H        ;INCREMENT PTR
007F   CD 0077 *      07800            CALL    $TTYOT   ;OUTPUT CHAR
0082   F1             07900            POP     PSW      ;GET COUNT
0083   3D             08000            DCR     A        ;DECREMENT IT
0084   C3 007B '      08100            JMP     DRV32    ;ONE MORE TIME
0087                  08200            END
```

```
        MAC80 1.0        PAGE    3

$IOERR  0011*   $BL     0043*   $BF     0048*   $ERR    003D*
$TTYIN  0018*   $TTYOT  0080*   IRECER  0012    $DRV3   0000'
DRV3FR  0013'   DRV3FW  0042'   DRV3BR  0010'   DRV3BW  0010'
DRV3RE  000E'   DRV3BA  000E'   DRV3EN  000E'   DRV31   0017'
DR3FW2  0079'   DR3FW1  0064'   DRV32   007B'
```

```
      MAC80 1.0        PAGE    1

                        00100   ;COMMENT *
                        00200   ;       DRIVER ADDRESSES FOR LUN'S 1 THROUGH 10
                        00210   ;
0001                    00220   LPT     EQU     1               ;UNIT 2 IS LPT
0001                    00230   DSK     EQU     1               ;UNITS 6-10 ARE DSK
0000                    00235   DTC     EQU     0               ;DTC COMMUNICATIONS UNIT
                        00240   ;
                        00300
0000                    00400           ENTRY   $LUNTB
0000                    00500           EXT     $DRV3
0000    0B              00600   $LUNTB: DB      013             ;MAX LUN + 1
0001    0000 *           00700          DW      $DRV3   ;THEY ALL POINT TO $DRV3 FOR NOW
0003                    00800           IFF     LPT
                        00900           DW      $DRV3
0003                    01000           ENDIF
0003                    01100           IFT     LPT
0003                    01200           EXT     LPTDRV
0003    0000 *          01300           DW      LPTDRV
0005                    01400           ENDIF
0005    0001 *          01500           DW      $DRV3
0007                    01510           IFF     DTC
```

```
0007    0005 *          01600           DW      $DRV3
0009                    01602           ENDIF
0009                    01604           IFT     DTC
                        01605           EXT     $CMDRV
                        01606           DW      $CMDRV
0009                    01608           ENDIF
0009    0007 *          01700           DW      $DRV3
000B                    01800           IFF     DSK
                        01900           DW      $DRV3
                        02000           DW      $DRV3
                        02100           DW      $DRV3
                        02200           DW      $DRV3
                        02300           DW      $DRV3
000B                    02400           ENDIF
000B                    02500           IFT     DSK
000B                    02600           EXT     DSKDRV
000B    0000 *          02700           DW      DSKDRV
000D    000B *          02800           DW      DSKDRV
000F    000D *          02900           DW      DSKDRV
0011    000F *          03000           DW      DSKDRV
0013    0011 *          03100           DW      DSKDRV
0015                    03200           ENDIF
0015                    03300           END
```

MAC80 1.0 PAGE 2

```
LPT     0001    DSK     0001    DTC     0000    $LUNTB  0000'
$DRV3   0009*   LPTDRV  0003*   DSKDRV  0013*
```

C/ Subprogram Linkages

This appendix defines a normal subprogram call as generated by the FORTRAN Compiler. It is included to facilitate linkages between FORTRAN programs and those written in other languages, such as Z8Ø Assembly.

A subprogram reference with no parameters generates a simple CALL instruction. The corresponding subprogram should return via a simple RET. (CALL and RET are Z8Ø opcodes.)

A subprogram reference with parameters results in a somewhat more complex calling sequence. Parameters are always passed by reference (that is, the thing passed is actually the address of the low bytes of the actual argument). Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends on the number of parameters to pass:

1. If the number of parameters is fewer than or equal to three, they are passed in the registers. Parameter 1 is in HL, 2 in DE (if present), and 3 in BC (if present).
2. If the number of parameters is more than 3, they are passed as follows:

    a. Parameter 1 in HL.
    b. Parameter 2 in DE.
    c. Parameters 3 through n in a contiguous data block. BC points to the low byte of this data block (that is, to the low byte of parameter 3).

Note that, with this scheme, the subprogram must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. Neither the Compiler nor the runtime system checks for the correct number of parameters.

If the subprogram expects more than three parameters and needs to transfer them to a local data area, a system subroutine performs this transfer.

This argument transfer routine is named $AT and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (that is, the total number of arguments minus 2). The subprogram is responsible for saving the first two parameters before calling $AT. For example, if a subprogram expects five parameters, it should appear as follows:

```
SUBR:     LD    (P1),HL    ;SAVE PARAMETER 1
          EX    DE,HL
          LD    (P2),HL    ;SAVE PARAMETER 2
          LD    A,3        ;NO. OF PARAMETERS LEFT
          LD    HL,P3      ;POINTER TO LOCAL AREA
          CALL  $AT        ;TRANSFER THE OTHER THREE PARAMETERS

          .
          .
          .
          {Body of subprogram}
          .
          .
          .
          RET              ;RETURN TO CALLER
P1:       DS    2          ;SPACE FOR PARAMETER 1
P2:       DS    2          ;SPACE FOR PARAMETER 2
P3:       DS    6          ;SPACE FOR PARAMETERS 3-5
```

When accessing parameters in a subprogram, don't forget that they are pointers to the actual arguments passed.

Important Note: You must see that the arguments in the calling program match in number, type, and length with the parameters the subprogram expects. This applies to FORTRAN subprograms and those written in assembly language.

FORTRAN functions return their values in registers and memory, depending on the type. Logical results are returned in (A), Integers in (HL), Reals in memory at $AC, and Double Precision in memory at $DAC. $AC and $DAC are the addresses of the low bytes of the mantissas.

D/ ASCII Character Codes

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
ØØØ	NUL	Ø43	+	Ø86	V
ØØ1	SOH	Ø44	,	Ø87	W
ØØ2	STX	Ø45	-	Ø88	X
ØØ3	ETX	Ø46	.	Ø89	Y
ØØ4	EOT	Ø47	/	Ø9Ø	Z
ØØ5	ENQ	Ø48	Ø	Ø91	[
ØØ6	ACK	Ø49	1	Ø92	/
ØØ7	BEL	Ø5Ø	2	Ø93	]
ØØ8	BS	Ø51	3	Ø94	^
ØØ9	HT	Ø52	4	Ø95	<
Ø1Ø	LF	Ø53	5	Ø96	'
Ø11	VT	Ø54	6	Ø97	a
Ø12	FF	Ø55	7	Ø98	b
Ø13	CR	Ø56	8	Ø99	c
Ø14	SO	Ø57	9	1ØØ	d
Ø15	SI	Ø58	:	1Ø1	e
Ø16	DLE	Ø59	;	1Ø2	f
Ø17	DC1	Ø6Ø	<	1Ø3	g
Ø18	DC2	Ø61	=	1Ø4	h
Ø19	DC3	Ø62	>	1Ø5	i
Ø2Ø	DC4	Ø63	?	1Ø6	j
Ø21	NAK	Ø64	@	1Ø7	k
Ø22	SYN	Ø65	A	1Ø8	l
Ø23	ETB	Ø66	B	1Ø9	m
Ø24	CAN	Ø67	C	11Ø	n
Ø25	EM	Ø68	D	111	o
Ø26	SUB	Ø69	E	112	p
Ø27	ESCAPE	Ø7Ø	F	113	q
Ø28	FS	Ø71	G	114	r
Ø29	GS	Ø72	H	115	s
Ø3Ø	RS	Ø73	I	116	t
Ø31	US	Ø74	J	117	u
Ø32	SPACE	Ø75	K	118	v
Ø33	!	Ø76	L	119	w
Ø34	"	Ø77	M	12Ø	x
Ø35	#	Ø78	N	121	y
Ø36	$	Ø79	O	122	z
Ø37	%	Ø8Ø	P	123	{
Ø38	&	Ø81	Q	124	
Ø39	'	Ø82	R	125	}
Ø4Ø	(	Ø83	S	126	
Ø41	)	Ø84	T	127	DEL
Ø42	*	Ø85	U		

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout

E/ FORLIB Arithmetic Library Subroutines

The FORTRAN-8Ø library contains a number of subroutines that you may reference from FORTRAN or assembly programs. In the following descriptions, $AC referes to the floating accumulator; $AC is the address of the low byte of the mantissa. $AC+3 is the address of the exponent. $DAC refers to the double precision accumulator; $DAC is the address of the low byte of the mantissa. $DAC+7 is the address of the double precision exponent.

All arithmetic routines (addition, subtraction, multiplication, division, exponentiation) adhere to the following calling conventions:

1. Argument 1 is passed in the registers:
   Integer in {HL}
   Real in $AC
   Double in $DAC
2. Argument 2 is passed either in registers or in memory, depending on the type:

   a. Integers are passed in {HL} or in {DE} if {HL} contains Argument 1.
   b. Real and double precision values are passed in memory pointed to by {HL}. ({HL} points to the low byte of the mantissa.)

The following arithemetic routines are contained in the Library:

Function	Name	Argument 1 Type	Argument 2 Type
Addition	$AA	Real	Integer
	$AB	Real	Real
	$AQ	Double	Integer
	$AR	Double	Real
	$AU	Double	Double

Division	$D9	Integer	Integer
	$DA	Real	Integer
	$DB	Real	Real
	$DQ	Double	Integer
	$DR	Double	Real
	$DU	Double	Double
Exponentiation	$E9	Integer	Integer
	$EA	Real	Integer
	$EB	Real	Real
	$EQ	Double	Integer
	$ER	Double	Real
	$EU	Double	Double
Multiplication	$M9	Integer	Integer
	$MA	Real	Integer
	$MB	Real	Real
	$MQ	Double	Integer
	$MR	Double	Real
	$MU	Double	Double
Subtraction	$SA	Real	Integer
	$SB	Real	Real
	$SQ	Double	Integer
	$SR	Double	Real
	$SU	Double	Double

Additional library routines are provided for converting between value types. Arguments are always passed to and returned by these conversion routines in the appropriate registers:

Logical in {A}
Integer in {HL}
Real in $AC
Double in $DAC

Name	Function
$CA	Integer to Real
$CC	Integer to Double
$CH	Real to Integer
$CJ	Real to Logical
$CK	Real to Double
$CX	Double to Integer
$CY	Double to Real
$CZ	Double to Logical

F/ Output File Format

Compilers and assemblers should ignore the line numbers and page marks included in the Editor output files (except when included in the listing files).

A line number consists of five decimal digits followed by a tab character.

A page mark is a line-feed character with the high-order bit equal to one.

G/ Storage Format

Type	Allocation
Integer	two bytes/ 1/2 storage unit S Binary Value Negative numbers are the two's complement of positive representations. The storage order is reversed. The least significant byte is followed by the most significant byte.
LOGICAL	one byte/ 1/4 storage unit Zero (false) or nonzero (true) A nonzero valued byte indicates true (the logical constant .TRUE. is represented by the hexadecimal value FF). A zero valued byte indicates false. When used as an arithmetic value, a Logical datum is treated as an Integer in the range -128 to +127.
REAL	four bytes/ one storage unit Characteristic S Mantissa (hi) Mantissa (mid) Mantissa (low)

The first byte is the characteristic expressed in excess 2ØØ (octal) notation; that is, a value of 2ØØ (octal) corresponds to a binary exponent of Ø. Values less than 2ØØ (octal) correspond to negative exponents, and values greater than 2ØØ correspond to positive exponents. By definition, if the characteristic is zero, the entire number is zero.

The next three bytes constitute the mantissa. The mantissa is always normalized such that the high order bit is one, eliminating the need to actually save that bit. The high bit is used instead to indicate a negative number, and zero indicates a positive number. The mantissa is assumed to be

a binary fraction of the binary point of which is to the left of the mantissa. The format of the mantissa is "signed magnitude." The bytes are stored in reverse order: mantissa low order, followed by mid order, high order, and characteristic.

DOUBLE PRECISION — **eight bytes**

The internal form of double precision data is identical with that of Real data except double precision uses four extra bytes for the mantissa.

INTEGER*4 — **four bytes**

Negative numbers of represented in two's complement form. The bytes are stored in reverse order, least significant to most significant.

H/ Format of Link-Compatible Object Files

Link-Compatible object files consist of a bit stream. A bit stream is the I/O "stream" of binary digits that represent data in coded form.

Use of a bit stream for relocatable object files keeps the size of the object files to a minimum, thereby decreasing the number of diskette read/writes you have to do.

Individual fields within the bit stream are not aligned on byte boundaries, except as noted below.

The two types of load items are Absolute (the actual addresses of information expressed in machine code) and Relocatable (a program coded in such a way that it can be stored and executed in any part of memory). The first bit of an item indicates one of these two types. If the first bit is a Ø, the following eight bits are loaded as an absolute byte. If the first bit is a 1, the next two bits are used to indicate one of four types of relocatable items:

ØØ	Special Linker item (see below).
Ø1	Program Relative. Loads the following 16 bits after adding the current Program base.
1Ø	1Ø Data Relative. Loads the following 16 bits after adding the current Data base.
11	Common Relative. Loads the following 16 bits after adding the current Common base.

Special Linker items consist of the bit stream 1ØØ followed by:

- a four-bit control field
- an optional A field consisting of a two-bit address type that is the same as the two-bit field above except ØØ specifies absolute address.

- an optional B field consisting of three bits that give a symbol length and up to eight bits for each character of the symbol.

A general representation of a special Linker item is:

```
1  ØØ xxxx   yy nn   zzz + characters of symbol name

          A field              B field
```

xxxx Four-bit control field (Ø-15 below)
yy Two-bit address type field
nn Sixteen-bit value
zzz Three-bit symbol length field

The following special types have a B-field only:

Ø Entry symbol (name for search)
1 Select COMMON block
2 Program name
3 Request library search
4 Reserved for future expansion

The following special Linker items have both an A field and a B field:

5 Define COMMON size
6 Chain external (A is head of address chain, B is name of external symbol)
7 Define entry point (A is address, B is name)
8 Reserved for future expansion

The following special Linker items have an A field only:

9 External +. offset. The A value will be added to the two bytes starting at the current location counter.
1Ø Define size of Data area (A is size)
11 Set loading location counter to A
12 Chain address. A is head of chain, replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.

13 Define program size (A is size)
14 End program (forces to byte boundary)

The following special Linker item has neither an A or a B field:

15 End file

I/ MACRO-8Ø Assembler

Assembly language programs and subroutines are assembled with MACRO-8Ø. Just as the FORTRAN compiler generates relocatable object code from a FORTRAN program, MACRO-8Ø generates relocatable object code from an assembly language program. Running MACRO-8Ø is very similar to running the FORTRAN compiler, and the command format is identical. The default extension for a MACRO-8Ø source file is /MAC.

1 Running MACRO-8Ø

When you give TRSDOS the command:

M8Ø

you are running the MACRO-8Ø assembler. When the assembler is ready to accept commands, it prompts you with an asterisk. To exit the assembler, use the <BREAK> key.

Command lines are also supported by MACRO-8Ø. After executing a command line, the assembler automatically exits to the operating system.

1.1 Command Format

An assembler command conveys the name of the source file you want to assemble and the options you want to use. Following is the format for an assembler command (square brackets indicate optional):

[object filename] [,listing filename]=source filename[-switch...]

Note: All filenames must be in TRSDOS filename format: filename[/ext] [.password] [:drive#]. If you are using the assembler's default extensions, it is not necessary to specify an extension in an assembler command.

Note each individual part of the assembler command:

1. Object filename
   To create a relocatable object file, this part of the command must be included. It is simply the name that you want to call the object file. The default extension for the object filename is /REL.

2. Listing filename
   To create a listing file, this part of the command must be included. It is simply the name that you want to call the listing file. The default extension for the listing file is /LST.

3. Source filename
   An assembler command must always include a source filename--this is the way the assembler "knows" the material to assemble. It is simply the name of a MACRO-8Ø program you have saved on disk. The default extension for a MACRO-8Ø source filename is /MAC. The source filename is always preceded by an equal sign in an assembler command.

Examples (asterisk is typed by M8Ø):

*=TEST	Assemble the program TEST/MAC without creating an object file or listing file.
*TEST,TEST=TEST	Assemble the program TEST/MAC. Create a relocatable object file called TEST/REL and a listing file called TEST/LST.
*,TEST.PASS=TEST.PASS	Assemble the program TEST/MAC.PASS and create a listing file called TEST/LST.PASS. (No object file is created.)

*TESTOBJ=TEST	Assemble the program TEST/MAC and create an object file called TESTOBJ/REL. (No listing file is created.)

4. Switch
A switch on the end of a command specifies a special parameter to be used during assembly. Switches are always preceded by a dash (-). More than one switch may be used in the same command. The available switches are:

Switch	Action
O	Print all listing addresses in octal.
H	Print all listing addresses in hexadecimal (default condition).
C	Force generation of a cross reference file.
Z	Assemble Z8Ø (Zilog format) mnemonics (default condition).
I	Assemble 8Ø8Ø mnemonics.

Examples:

*CT.ME,CT.ME=CT.ME-O	Assemble the program CT/MAC.ME. Create a listing file called CT/LST.ME and an object file called CT/REL.ME. The addresses in the listing file will be in octal.

*LT,LT=LT-C	Assemble the program LT/MAC. Create an object file called LT/REL, a listing file called LT/LST, and a cross reference file called LT/CRF. (See Section 12.)

1.2 Linkage to FORTRAN Programs

To link an assembly language subroutine to a FORTRAN program, use the following format:

```
L8Ø
*PROG,MYASM,PROG-N-E
```

In this example, MYASM is the name of the assembly language subroutine, and PROG/REL is the name of the FORTRAN program. MYASM/REL cannot be assembled with an END <label> statement.

2 Format of MACRO-8Ø Source Files

In general, MACRO-8Ø accepts a source file that is almost identical to source files for INTEL compatible assemblers. Input source lines of up to 132 characters in length are acceptable.

MACRO-8Ø preserves lowercase letters in quoted strings and comments. All symbols, opcodes, and pseudo-opcodes typed in lowercase will be converted to uppercase.

Note: If the source file includes line numbers from an editor, each byte of the line number must have the high bit on. Line numbers from Microsoft's EDIT-8Ø Editor are acceptable.

2.1 Statements

Source files input to MACRO-8Ø consist of statements of the form:

[label:[:]] [operator] [arguments] [;comment]

With the exception of the ISIS assembler $ controls, it is not necessary that statements begin in Column 1. Multiple blanks or tabs may be used to improve readability.

If a label is present, it is the first item in the statement and is immediately followed by a colon. If it is followed by two colons, it is declared as PUBLIC. (See ENTRY/PUBLIC, Section 5.1Ø.) For example:

```
FOO::    RET
```

is equivalent to:

```
PUBLIC   FOO
FOO:     RET
```

The next item after the label (or the first item on the line i, no label is present) is an operator. An operator may be an opcode (8Ø8Ø or Z8Ø mnemonic), pseudo-op, macro call or expression. The evaluation order is:

1. Macro call

2. Opcode/Pseudo operation

3. Expression

Instead of flagging.an expression as an error, the assembler treats it as if it were a DB statement. (See Section 5.4.)

The arguments following the operator will, of course, vary in form according to the operator.

A comment always begins with a semicolon and ends with a carriage return. A comment may be a line by itself, or it may be appended to a line that contains a statement. Extended comments can be entered using the .COMMENT pseudo operation. (See Section 5.19.)

2.2 Symbols

MACRO-8Ø symbols may be of any length; however, only the first six characters are significant. The following characters are legal in a symbol:

A-Z Ø-9 $. ? @

The underline character is also legal in a symbol. A symbol cannot start with a digit. When a symbol is read, lowercase is translated into uppercase. If a symbol reference is followed by ##, it is declared external. (See also the EXT/EXTRN pseudo-op, Section 5.12.)

2.3 Numeric Constants

The default base for numeric constants is decimal. This may be changed by the .RADIX pseudo-op. (See Section 5.21.) Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the base is greater than 1Ø, A-F are the digits following 9. If the first digit of the number is not numeric (such as A-f), the number must be preceded by a zero. **This eliminates the use of zero as a leading digit for octal constants, as in previous versions of MACRO-8Ø.**

Numbers are 16-bit unsigned quantities. A number is always evaluated in the current radix unless you use one of the following special notations:

nnnnB	Binary
nnnnD	Decimal
nnnnO	Octal
nnnnQ	Octal
nnnnH	Hexadecimal
X'nnnn'	Hexadecimal

Overflow of a number beyond two bytes is ignored, and the result is the low order 16-bits.

A character constant is a string comprised of zero and one or two ASCII characters, delimited by quotation marks and used in a non-simple expression. For example, in the statement:

```
DB      'A' + 1
```

'A' is a character constant. However, the statement:

```
DB      'A'
```

uses 'A' as a string because it is in a simple expression. The rules for character constant delimiters are the same as for strings.

A character constant comprised of one character has as its value the ASCII value of that character. That is, the high order byte of the value is zero, and the low order byte is the ASCII value of the character. For example, the value of the constant, 'A', is 41H.

A character constant comprised of two characters has as its value the ASCII value of the first character in the high order byte and the ASCII value of the second character in the low order byte. For example, the value of the character constant, "AB", is 41H*256+42H.

2.4 Strings

A string is comprised of zero or more characters delimited by quotation marks. Either single or double quotation marks may be used as string delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement:

```
DB      "I am ""great"" today"
```

stores the string:

```
I am "great" today
```

If there are zero characters between the delimiters, the string is a null string.

3 Expression Evaluation

3.1 Arithmetic and Logical Operators

The following operators are allowed in expressions. The operators are listed in order of precedence.

```
NUL
LOW, HIGH
*, /, MOD, SHR, SHL
Unary Minus
+, -
EQ, NE, LT, LE, GT, GE
NOT
AND
OR, XOR
```

Parentheses are used to change the order of precedence. During evaluation of an expression, as soon as a new operator is encountered having precedence less than or equal to the last operator encountered, all operations up to the new operator are performed. That is, subexpressions involving operators of higher precedence are computed first.

All operators except +, -, *, and / must be separated from their operands by at least one space.

The byte isolation operators (HIGH, LOW) isolate the high or low order 8 bits of an Absolute 16-bit value. If a relocatable value is supplied as an operand, HIGH and LOW treat it as if it were relative to location zero.

3.2 Modes

All symbols used as operands in expressions are in one of the following modes: Absolute, Data Relative, Program (Code) Relative, or COMMON. (See Section 5 for the ASEG, CSEG, DSEG, and COMMON pseudo-ops.) Symbols assembled under the ASEG, CSEG (default), or DSEG pseudo-ops are in

Absolute, Code Relative, or Data Relative modes, respectively. The number of COMMON modes in a program is determined by the number of COMMON blocks that are named with the COMMON pseudo-op. Two COMMON symbols are not in the same mode unless they are in the same COMMON block.

In any operation other than addition or subtraction, the mode of both operands must be Absolute.

If the operation is addition, the following rules apply:

1. At least one of the operands must be Absolute.

2. Absolute + <mode> = <mode>

If the operation is subtraction, the following rules apply:

1. <mode> - Absolute = <mode>

2. <mode> - <mode> = Absolute, where the two <mode>s are the same.

Each intermediate step in the evaluation of an expression must conform to the above rules for modes; otherwise, an error is generated. For example, if FOO, BAZ, and ZAZ are three Program Relative symbols, the expression:

```
FOO + BAZ - ZAZ
```

generates an R error because the first step (FOO + BAZ) adds two relocatable values. (One of the values must be Absolute.) This problem can always be remedied by inserting parentheses. The expression:

```
FOO + (BAZ - ZAZ)
```

is legal because the first step (BAZ - ZAZ) generates an Absolute value that is then added to the Program Relative value, FOO.

3.3 Externals

Aside from its classification by mode, a symbol is either External or not External. (See EXT/EXTRN, Section 5.12.) An External value must be assembled into a two-byte field. (Single-byte Externals are not supported.) The following rules apply to the use of Externals in expressions.

1. Externals are legal only in addition and subtraction.
2. If an External symbol is used in an expression, the result of the expression is always External.
3. When the operation is addition, either operand (but not both) may be External.
4. When the operation is subtraction, only the first operand may be External.

4 Opcodes as Operands

8Ø8Ø opcodes are valid one-byte operands. Note that only the first byte is a valid operand. For example:

```
MVI     A,(JMP)
ADI     (CPI)
MVI     B, (RNZ)
CPI     (INX H)
ACI     (LXI B)
MVI     C,MOV A,B
```

Errors are generated if more than one byte is included in the operand--such as (CPI 5), (LXI B,LABEL1), or (JMP LABEL2).

Opcodes used as one-byte operands need not be enclosed in parentheses.

Note: Opcodes are not valid operands in Z8Ø mode.

5 Pseudo Operations

5.1 ASEG

ASEG

ASEG sets the location counter to an absolute segment of memory. The location of the absolute counter will be that of the last ASEG to change the location. The effect of ASEG is also achieved by using the code segment (CSEG) pseudo operation and the -P switch in LINK-8Ø. See also Section 5.27.

5.2 COMMON

COMMON /<block name>/

COMMON sets the location counter to the selected common block in memory. The location is always the beginning of the area so that compatibility with the FORTRAN COMMON statement is maintained. If <block name> is omitted or consists of spaces, it is considered to be blank common. See also Section 5.27.

5.3 CSEG

CSEG

CSEG sets the location counter to the code relative segment of memory. The location will be that of the last CSEG (default is Ø), unless an ORG is done after the CSEG to change the location. CSEG is the default condition of the assembler. (The INTEL assembler defaults to ASEG.) See also Section 5.27.

5.4 Define Byte

```
DB      <exp>[,<exp>...]

DB      <string>[<string>...]
```

The arguments to DB are either expressions or strings. DB stores the values of the expressions or the characters of the strings in successive memory locations, beginning with the current location counter.

Expressions must evaluate to one byte. (If the high byte of the result is Ø or 255, no error is given; otherwise, an A error results.)

Strings of three or more characters may not be used in expressions. (That is, they must be immediately followed by a comma or by the end of the line.) The characters in a string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

Example:

```
ØØØØ'    4142          DB      'AB'
ØØØ2'    42            DB      'AB' AND ØFFH
ØØØ3'    41 42 43      DB      'ABC'
```

5.5 Define Character

```
DC      <string>
```

DC stores the characters in <string> in successive memory locations, beginning with the current location counter. As with DB, characters are stored in order of appearance, each as a one-byte value with the high order bit set to zero. However, DC stores the last character of the string with the high bit set to one. An error results is the argument to DC is a null string.

5.6 Define Space

```
DS      <exp>
```

DS reserves an area of memory. The value of <exp> gives the number of bytes to be allocated. All names used in <exp> must be previously defined (that is, all names known at that point on pass 1). Otherwise, a V error is generated during pass 1, and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the DS generated no code on pass 1.

5.7 DSEG

```
DSEG
```

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter will be that of the last DSEG (default is Ø), unless an ORG is done after the DSEG to change the location. See also Section 5.27.

5.8 Define Word

```
DW      <exp>[,<exp>...]
```

DW stores the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values.

5.9 END

```
END     [<exp>]
```

The END statement specifies the end of the program. If <exp> is present, it is the start address of the program.

If <exp> is not present, then no start address is passed to LINK-8Ø for that program.

5.1Ø ENTRY/PUBLIC

```
ENTRY <name>[,<name>...]
```

or

```
PUBLIC <name>[,<name>...]
```

ENTRY or PUBLIC declares each name in the list as internal and therefore available for use by this program and other programs to be loaded concurrently. All the names in the list must be defined in the current program; otherwise, a U error results. An M error is generated if the name is an external name or common-blockname.

5.11 EQU

```
<name> EQU <exp>
```

EQU assigns the value of <exp> to <name>. If <exp> is external, an error is generated. If <name> already has a value other than <exp>, an M error is generated.

5.12 EXT/EXTRN

```
EXT <name>[,<name>...]
```

or

```
EXTRN <name>[,<name>...]
```

EXT or EXTRN declares that the name(s) in the list are external (that is, defined in a different program). If any item in the list references a name that is defined in the current program, an M error results. A reference to a name in which the name is immediately followed by two pound signs (such as NAME##) also declares the name as external.

5.13 NAME

```
NAME ('modname')
```

NAME defines a name for the module. Only the first six characters are significant in a module name. A module name may also be defined with the TITLE pseudo-op. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source file name.

5.14 Define Origin

```
ORG <exp>
```

The location counter is set to the value of <exp>, and the assembler assigns generated code starting with that value. All names used in <exp> must be known on pass 1, and the value must either be absolute or in the same area as the location counter.

5.15 PAGE

```
PAGE [<exp>]
```

PAGE causes the assembler to start a new output page. The value of <exp>, if included, becomes the new page size (measured in lines per page) and must be in the range 1Ø-255. The default page size is 5Ø lines per page. The assembler puts a form feed character in the listing file at the end of a page.

5.16 SET

```
<name> SET <exp>
```

SET is the same as EQU except that no error is generated if <name> is already defined.

5.17 SUBTTL

SUBTTL <text>

SUBTTL specifies a subtitle to be listed on the line after the title on each page heading. (See TITLE, Section 5.18.) <text> is truncated after 60 characters. Any number of SUBTTLs may be given in a program.

5.18 TITLE

TITLE <text>

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results. The first six characters of the title are used as the module name unless a NAME pseudo operation is used. If neither a NAME nor TITLE pseudo-op is used, the module name is created from the source file name.

5.19 .COMMENT

.COMMENT <delim><text><delim>

The first non-blank character encountered after .COMMENT is the delimiter. The following <text> comprises a comment block that continues until the next occurrence of <delimiter> is encountered. For example, using an asterisk as the delimiter, the format of the comment block would be:

```
.COMMENT  *
any amount of text entered here as the comment
  block
  .
  .
  .       *
;return to normal mode
```

5.2Ø .PRINTX

```
.PRINTX <delim><text><delim>
```

The first non-blank character encountered after .PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered. .PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches. For example:

```
IF      CPM
.PRINTX /CPM version/
ENDIF
```

Note: .PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op.

5.21 .RADIX

```
.RADIX <exp>
```

The default base (or radix) for all constants is decimal. The .RADIX statement allows the default radix to be changed to any base in the range 2-16. For example:

```
LXI     H,ØFFH
.RADIX 16
LXI     H,ØFF
```

The two LXIs in the example are identical. The <exp> in a .RADIX statement is always in decimal radix, regardless of the current radix.

5.22 .REQUEST

.REQUEST <filename>[,<filename>...]

.REQUEST sends a request to the LINK-8Ø loader to search the filenames in the list for undefined globals before searching the FORTRAN library. The filenames in the list should be in the form of legal MACRO-8Ø symbols. They sould not include filename extensions or disk specifications. The LINK-8Ø loader supplies its default extension and assumes the currently selected disk drive.

5.23 .Z8Ø

.Z8Ø enables the assembler to accept Z8Ø opcodes. This is the default condition. Z8Ø mode may also be set by appending the Z switch to the MACRO-8Ø command string--see Section 1.2.

5.24 .8Ø8Ø

.8Ø8Ø enables the assembler to accept 8Ø8Ø opcodes. 8Ø8Ø mode may also be set by appending the I switch to the MACRO-8Ø command string--see Section 1.2.

5.25 Conditional Pseudo Operations

The conditional pseudo operations are:

IF/IFT <exp>	True if <exp> is not Ø.
IFE/IFF <exp>	True if <exp> is Ø.
IF1	True if pass 1.
IF2	True if pass 2.
IFDEF <symbol>	True if <symbol> is defined or has been declared External.

IFNDEF <symbol>	True if <symbol> is undefined or not declared External.
IFB <arg>	True if <arg> is blank. The angle brackets around <arg> are required.
IFNB <arg>	True if <arg> is not blank. Used for testing when dummy parameters are supplied. The angle brackets around <arg> are required.

All conditionals use the following format:

```
IFxx    [argument]
  .
  .
  .
[ELSE
  .
  .
  .     ]
ENDIF
```

Conditionals may be nested to any level. Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF, IFT, IFF, and IFE, the expression must involve values that were previously defined, and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it is defined on pass 2.

ELSE

Each conditional pseudo operation may optionally be used with the ELSE pseudo operation that allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF, and an ELSE is always bound to the most recent, open IF. A conditional with more thanone ELSE or an ELSE without a conditional causes a C error.

ENDIF
Each IF must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a C error.

5.26 Listing Control Pseudo Operations

Output to the listing file can be controlled by two pseudo-ops:

.LIST and .XLIST

If a listing is not being made, these pseudo-ops have no effect. .LIST is the default condition. When a .XLIST is encountered, source and object code is not listed until a .LIST is encountered.

The output of cross reference information is controlled by .CREF and .XCREF. If the cross reference facility (Section 12) has not been invoked, .CREF and .XCREF have no effect. The default condition is .CREF. When a .XCREF is encountered, no cross reference information is output until .CREF is encountered.

The output of MACRO/REPT/IRP/IRPC expansions is controlled by three pseudo-ops: .LALL, .SALL, and .XALL. .LALL lists the complete macro text for all exapnsions. .SALL lists only the object code produced by a macro and not its text. .XALL is the default condition; it is similar to .SALL except that a source line is listed only if it generates object code.

5.27 Relocation Pseudo Operations

The ability to create relocatable modules is one of the major features of MACRO-8Ø. Relocatable modules offer the advantages of easier coding and faster testing, debugging, and modifying. In addition, it is possible to specify segments of assembled code that will later be

loaded into RAM (the Data Relative segment) and ROM/PROM (the Code Relative segment). The pseudo operations that select relocatable areas are CSEG and DSEG. The ASEG pseudo-op is used to generate non-relocatable (absolute) code. The COMMON pseudo-op creates a common data area for every COMMON block named in the program.

The default mode for the assembler is Code Relative. That is, assembly begins with a CSEG automatically executed and the location counter in the Code Relative mode, pointing to location Ø in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG or DSEG or COMMON pseudo-op is executed. For example, the first DSEG encountered sets the location counter to location Ø in the Data Relative segment of memory. The following code is assembled in the Data Relative mode, that is, it is assigned to the Data Relative segment of memory. If a subsequent CSEG is encountered, the location counter returns to the next free location in the Code Relative segment, and so on.

The ASEG, DSEG, and CSEG pseudo-ops never have operands. If you wish to alter the current value of the location counter, use the ORG pseudo-op.

ORG Pseudo-op

At any time, the value of the location counter may be changed by use of the ORG pseudo-op. The form of the ORG statement is:

```
ORG     <exp>
```

where the value of <exp> will be the new value of the location counter in the current mode. All names used in <exp> must be known on pass 1, and the value of <exp> must be either Absolute or in the current mode of the location counter. For example, the statements:

```
DSEG
ORG     5Ø
```

set the Data Relative location counter to 5Ø, relative to the start of the Data Relative segment of memory.

LINK-8Ø

The LINK-8Ø linking loader (Chapter 4 of this manual) combines the segments and creates each relocatable module in memory when the program is loaded. The origins of the relocatable segments are not fixed until the program is loaded, and the origins are assigned by LINK-8Ø. The command to LINK-8Ø may contain user-specified origins through the use of the -P (for Code Relative) and -D (for Data and COMMON segments) switches.

For example, a program that begins with the statements:

```
ASEG
ORG     8ØØH
```

and is assembled entirely in Absolute mode always loads beginning at 8ØØ unless the ORG statement is changed in the source file. However, the same program, assembled in Code Relative mode with no ORG statement, may be loaded at any specified address by appending the -P:<address> switch to the LINK-8Ø command string.

5.28 Relocation Before Loading

Two pseudo-ops, .PHASE and .DEPHASE, allow code to be located in one area but executed only at a different, specified area.

For example:

```
ØØØØ'                                  .PHASE      1ØØH
Ø1ØØ      CD Ø1Ø6       FOO:           CALL        BAZ
Ø1Ø3      C3 ØØØ7'                     JMP         ZOO
Ø1Ø6      C9            BAZ:           RET
                                       .DEPHASE
ØØØ7'     C3 ØØØ5       ZOO:           JMP         5
```

All labels within a .PHASE block are defined as the absolute value from the origin of the phase area. The code, however, is loaded in the current area (that is, from Ø' in this example). The code within the block can later be moved to 1ØØH and executed.

6 Macros and Block Pseudo Operations

The macro facilities provided by MACRO-8Ø include three repeat pseudo operations: repeat (REPT), indefinite repeat (IRP), and indefinite repeat character (IRPC). A macro definition operation (MACRO) is also provided. Each of these four macro operations is terminated by the ENDM pseudo operation.

6.1 Terms

For the purpose of discussion of macros and block operations, the following terms will be used:

1. <dummy> is used to represent a dummy parameter. All dummy parameters are legal symbols that appear in the body of a macro expansion.

2. <dummylist> is a list of <dummy>s separated by commas.

3. <arglist> is a list of arguments separated by commas. <arglist> must be delimited by angle brackets. Two angle brackets with no intervening characters (<>) or two commas with no intervening characters enter a null argument in the list. Otherwise, an argument is a character or series of characters terminated by a comma or >. With angle brackets that are nested inside an <arglist>, one level of brackets is removed each time the bracketed argument is used in an <arglist>. (See the example in Section 6.5.) A quoted string is an acceptable argument and is passed as such. Unless enclosed in brackets or a quoted string, leading and trailing spaces are deleted from arguments.

4. <paramlist> is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is terminated by the end of line or a comment), but the rules for entering null parameters and nesting brackets are the same as

described for <arglist>. (See the example in Section 6.5.)

6.2 REPT-ENDM

```
        REPT    <exp>
         .
         .
         .
        ENDM
```

The block of statements between REPT and ENDM is repeated <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains any external or undefined terms, an error is generated. Example:

```
X       SET     Ø
        REPT    1Ø      ;generates DB1-DB1Ø
X       SET     X+1
        DB      X
        ENDM
```

6.3 IRP-ENDM

```
        IRP     <dummy>,<arglist>
         .
         .
         .
        ENDM
```

The <arglist> must be enclosed in angle brackets. The number of arguments in the <arglist> determines the number of times the block of statements is repeated. Each repetition substitutes the next item in the <arglist> for every occurrence of <dummy> in the block. If the <arglist> is null (that is, <>), the block is processed once with each occurrence of <dummy> removed. For example:

```
IRP     X,<1,2,3,4,5,6,7,8,9,1Ø>
DB      X
ENDM
```

generates the same bytes as the REPT example.

6.4 IRPC-ENDM

```
IRPC    <dummy>,string (or <string>)
 .
 .
 .
ENDM
```

IRPC is similar to IRP, but the arglist is replaced by a string of text, and the angle brackets around the string are optional. The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block. For example:

```
IRPC    X,Ø123456789
DB      X+1
ENDM
```

generates the same code as the two previous examples.

6.5 MACRO

Often, it is convenient to be able to generate a given sequence of statements from various places in a program, even though different parameters may be required each time the sequence is used. This capability is provided by the MACRO statement. The form is:

```
<name> MACRO <dummylist>
          .
          .
          .
       ENDM
```

where <name> conforms to the rules for forming symbols. <name> is the name that will be used to invoke the macro. The <dummy>s in <dummylist> are the parameters that will be changed (replaced) each time the MACRO is invoked. The statements before the ENDM comprise the body of the macro. During assembly, the macro is expanded every time it is invoked, but, unlike REPT/IRP/IRPC, the macro is not expanded when it is encountered.

The form of a macro call is:

```
<name> <paramlist>
```

where <name> is the name supplied in the MACRO definition, and the parameters in <paramlist> will replace the <dummy>s in the MACRO <dummylist> on a one-to-one basis. The number of items in <dummylist> and <paramlist> is limited only by the length of a line. The number of parameters used when the macro is called need not be the same as the number of <dummy>s in <dummylist>. If there are more parameters than <dummy>s, the extras are ignored. If there are fewer, the extra <dummy>s are made null. The assembled code contains the macro expansion code after each macro call.

Note: A dummy parameter in a MACRO/REPT/IRP/IRPC is always recognized exclusively as a dummy parameter. Register names, such as A and B, are changed in the expansion if they were used as dummy parameters.

Following is an example of a MACRO definition that defines a macro called FOO:

```
FOO     MACRO   X
Y       SET     Ø
        REPT    X
Y       SET     Y+1
        DB      Y
        ENDM
        ENDM
```

This macro generates the same code as the previous three examples when the call:

```
FOO    1Ø
```

is executed.

Another example, which generates the same code, illustrates the removal of one level of brackets when an argument is used as an arglist:

```
FOO    MACRO  X
       IRP    Y,<X>
       DB     Y
       ENDM
       ENDM
```

When the call:

```
FOO    <1,2,3,4,5,6,7,8,9,1Ø>
```

is made, the macro expansion looks like this:

```
IRP    Y,<1,2,3,4,5,6,7,8,9,1Ø>
DB     Y
ENDM
```

6.6 ENDM

Every REPT, IRP, IRPC, and MACRO pseudo-op must be terminated with the ENDM pseudo-op. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM causes an O error.

6.7 EXITM

The EXITM pseudo-op is used to terminate a REPT/IRP/IRPC or MACRO call. When an EXITM is executed, the expansion is exited immediately, and any remaining expansion or

repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

6.8 LOCAL

```
LOCAL   <dummylist>
```

The LOCAL pseudo-op is allowed only inside a MACRO definition. When LOCAL is executed, the assembler creates a unique symbol for each <dummy> in <dummylist> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiply-defined labels on successive expansions of the macro. The symbols created by the assembler range from ..ØØØ1 to ..FFFF. You will therefore want to avoid the form ..nnnn for your own symbols. If LOCAL statements are used, they must be the first statements in the macro definition.

6.9 Special Macro Operators and Forms

& The ampersand is used in a macro expansion to concatenate text or symbols. A dummy parameter that is in a quoted string will not be substituted in the expansion unless it is immediately preceded by &. To form a symbol from text and a dummy, put & between them. For example:

```
ERRGEN  MACRO   X
ERROR&X:PUSH    B
        MVI     B,'&X'
        JMP     ERROR
        ENDM
```

In this example, the call ERRGEN A generates:

```
ERRORA: PUSH    B
        MVI     B,'A'
        JMP     ERROR
```

;; In a block operation, a comment preceded by two semicolons is not saved as part of the expansion (that is, will not appear on the listing, even under.LALL). A comment preceded by one semicolon, however, is preserved and appears in the expansion.

! When an exclamation point is used in an argument, the next character is entered literally. (That is, !; and <;> are equivalent.)

NUL NUL is an operator that returns true if its argument (a parameter) is null. The remainder of a line after NUL is considered to be the argument to NUL. The conditional:

```
IF NUL argument
```

is false if, during the expansion, the first character of the argument is anything other than a semicolon or carriage return. We recommend that you test for null parameters using the IFB and IFNB conditionals.

7 Using Z8Ø Pseudo-ops

The following Z8Ø speudo ops are valid. The function of each pseudo-op is equivalent to that of its 8Ø8Ø counterpart.

Z8Ø Pseudo-op	Equivalent 8Ø8Ø Pseudo-op
COND	IFT
ENDC	ENDIF
*EJECT	PAGE
DEFB	DB
DEFS	DS
DEFW	DW
DEFM	DB
DEFL	SET
GLOBAL	PUBLIC
EXTERNAL	EXTRN

The formats, where different, conform to the 8Ø8Ø format. That is, DEFB and DEFW are permitted a list of arguments (as are DB and DW), and DEFM is permitted a string or numeric argument (as is DB).

8 Sample Assembly

```
TRSDOS READY
M8Ø
*EXMPL1,TTY=EXMPL1

        MAC8Ø 3.44        PAGE    1


                                    ØØ1ØØ   ;CSL3(P1,P2)
                                    ØØ2ØØ   ;SHIFT P1 LEFT CIRCULARLY 3
                                    ØØ3ØØ   ;RETURN RESULT IN P2
                                    ØØ4ØØ           ENTRY   CSL3
                                    ØØ45Ø   ;GET VALUE OF FIRST PARAMET
                                    ØØ5ØØ   CSL3:
ØØØØ'   7E                          ØØ6ØØ           MOV     A,M
ØØØ1'   23                          ØØ7ØØ           INX     H
ØØØ2'   66                          ØØ8ØØ           MOV     H,M
ØØØ3'   6F                          ØØ9ØØ           MOV     L,A
                                    Ø1ØØØ   ;SHIFT COUNT
ØØØ4'   Ø6 Ø3                       Ø11ØØ           MVI     B,3
ØØØ6'   AF                          Ø12ØØ   LOOP:   XRA     A
                                    Ø13ØØ   ;SHIFT LEFT
ØØØ7'   29                          Ø14ØØ           DAD     H
                                    Ø15ØØ   ;ROTATE IN CY BIT
ØØØ8'   17                          Ø16ØØ           RAL
ØØØ9'   85                          Ø17ØØ           ADD     L
ØØØA'   6F                          Ø18ØØ           MOV     L,A
                                    Ø19ØØ   ;DECREMENT COUNT
ØØØB'   Ø5                          Ø2ØØØ           DCR     B
                                    Ø21ØØ   ;ONE MORE TIME
ØØØC'   C2 ØØØ6'                    Ø22ØØ           JNZ     LOOP
ØØØF'   EB                          Ø23ØØ           XCHG
                                    Ø24ØØ   ;SAVE RESULT IN SECOND PARAI
ØØ1Ø'   73                          Ø25ØØ           MOV     M,E
ØØ11'   23                          Ø26ØØ           INX     H
ØØ12'   72                          Ø27ØØ           MOV     M,D
ØØ13'   C9                          Ø28ØØ           RET
                                    Ø29ØØ           END
```

```
        MAC8Ø 3.44        PAGE      S

CSL3    ØØØØI'  LOOP      ØØØ6'

No Fatal error(s)
```

Note: Use the -I switch if you assemble this routine.

9 MACRO-8Ø Errors

MACRO-8Ø errors are indicated by a one-character flag in column one of the listing file. If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal. Following is a list of the MACRO-8Ø error codes.

A Argument error
Argument to pseudo-op is not in correct format or is out of range (.PAGE 1; .RADIX 1; PUBLIC 1; STAX H; MOV M,M; INX C).

C Conditional nesting error
ELSE without IF, ENDIF without IF, two ELSEs on one IF.

D Double defined symbol
Reference to a symbol that is multiply defined.

E External error
Use of an external illegal in context (for example, FOO SET NAME##; MVI A,2-NAME##).

M Multiply defined symbol
Definition of a symbol that is multiply defined.

N Number error
Error in a number, usually a bad digit (for example, 8Q).

O Bad opcode or objectionable syntax
ENDM, LOCAL outside a block; SET, EQU, or MACRO without a name; bad syntax in an opcode (MOV A:); or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, and so on).

P Phase error
Value of a label or EQU name is different on pass 2.

Q Questionable
Usually means that a line is not terminated properly. This is a warning error (for example, MOV A,B,).

R Relocation
Illegal use of relocation in expression, such as abs-rel. Data, code, and COMMON areas are relocatable.

U Undefined symbol
A symbol referenced in an expression is not defined. (For certain pseudo-ops, a V error is printed on pass 1 and a U on pass 2.)

V Value error
On pass 1, a pseudo-op that must have its value known on pass 1 (for example, .RADIX, .PAGE, DS, IF, IFE, and so on) has a value that is undefined. If the symbol is defined later in the program, a U error will not appear on the pass 2 listing.

Error Messages:

'No end statement encountered on input file'
No END statement; either it is missing, or it is not parsed due to being in a false conditional, unterminated IRP/IRPC/REPT block or terminated macro.

'Unterminated conditional'
At least one conditional is unterminated at the end of the file.

'Unterminated REPT/IRP/IRPC/MACRO'
At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]
The number of fatal errors and warnings. The message is listed on the CRT and in the list file.

1Ø Compatibility with Other Assemblers

The $EJECT and $TITLE controls are provided for compatibility with INTEL's ISIS assembler. The dollar sign must appear in Column 1 only if spaces or tabs separate the dollar sign from the control word. The control:

```
$EJECT
```

is the same as the MACRO-8Ø PAGE pseudo-op. The control:

```
$TITLE('text')
```

is the same as the MACRO-8Ø SUBTTL <text> pseudo-op.

The INTEL operands, PAGE and INPAGE, generate Q errors when used with the MACRO-8Ø CSEG or DSEG pseudo-ops. These errors are warnings; the assembler ignores the operands.

When you enter MACRO-8Ø, the default for the origin is Code Relative Ø. With the INTEL ISIS assembler, the default is Absolute Ø.

With MACRO-8Ø, the dollar sign ($) is a defined constant that indicates the value of the location counter at the start of the statement. Other assemblers may use a decimal point or an asterisk. Other constants are defined by MACRO-8Ø to have the following values:

```
A=7    B=Ø    C=1    D=2    E=3
H=4    L=5    M=6   SP=6  PSW=6
```

11 Format of Listings

On each page of a MACRO-8Ø listing, the first two lines have the form:

```
[TITLE text]    MAC8Ø 3.2    PAGE x[-y]
[SUBTTL text]
```

where:

1. TITLE text is the text supplied with the TITLE pseudo-op, if one was given in the source program.

2. x is the major page number, which is incremented only when a form feed is encountered in the source file. (When using Microsoft's EDIT-8Ø text editor, a form feed is inserted whenever a page mark is done.) When the symbol table is being printed, x = 'S'.

3. y is the minor page number, which is incremented whenever the .PAGE pseudo-op is encountered in the source file, or whenever the current page size has been filled.

4. SUBTTL text is the text supplied with the SUBTTL pseudo-op if one was given in the source program.

Next, a blank line is printed, followed by the first line of output.

A line of output on a MACRO-8Ø listing has the following form:

```
[crf#]    [error] loc#m    xx    xxxx ...    source
```

If cross reference information is being output, the first item on the line is the cross reference number, followed by a tab.

A one-letter error code followed by a space appears next on the line if the line contains an error. If there is no error, a space is printed. If there is no cross

reference number, the error code column is the first column on the listing.

The value of the location counter appears next on the line. It is a four-digit hexadecimal number or six-digit octal number, depending upon whether the -O or -H switch was given in the MACRO-8Ø command string.

The character at the end of the location counter value is the mode indicator. It is one of the following symbols:

'	Code Relative
"	Data Relative
!	COMMON Relative
<space>	Absolute
*	External

Next, three spaces are printed, followed by the assembled code. One-byte values are followed by a space. Two-byte values are followed by a mode indicator. Two-byte values are printed in the opposite order from the order in which they are stored, that is, the high order byte is printed first. Externals are either the offset or the value of the pointer to the next External in the chain.

The remainder of the line contains the line of source code as it was input.

11.1 Symbol Table Listing

In the symbol table listing, all the macro names in the program are listed alphabetically, followed by all the symbols in the program, listed alphabetically. After each symbol, a tab is printed, followed by the value of the symbol. If the symbol is Public, an I prints immediately after that value. The next character printed is one of the following:

U Undefined symbol.

C	COMMON block name. (The "value" of the COMMON block is its length, or number of bytes, in hexadecimal or octal.)
*	External symbol.
<space>	Absolute value.
'	Program Relative value.
"	Data Relative value.
!	COMMON Relative value.

12 Cross Reference Facility

The Cross Reference Facility is invoked by typing CREF8Ø at TRSDOS command level. To generate a cross reference listing, the assembler must output a special listing file with embedded control characters. The MACRO-8Ø command string tells the assembler to output this special listing file. (See Section 5.26 for the .CREF and .XCREF pseudo-ops.) -C is the cross reference switch. When the -C switch is encountered in a MACRO-8Ø command string, the assembler opens a /CRF file instead of a /LST file.

Examples:

*,TEST=TEST-C	Assemble the file TEST/MAC and create cross reference file TEST/CRF.
*T,U=TEST-C	Assemble file TEST/MAC and create object file T/REL and cross reference file U/CRF.

When the assembler is finished, you must call the cross reference facility by typing CREF8Ø. (CREF8Ø is on Diskette #1). CREF8Ø command format is:

*listing file=source file

The default extension for the source file is /CRF. The -L switch is ignored, and any other switch causes an error message to be sent to the terminal.

Example:

*T=TEST	Examine fie TEST/CRF and generate a cross reference listing file T/LST.

Cross reference listing files differ from ordinary listing files in that:

1. Each source statement is numbered with a cross reference number.

2. At the end of the listing, variable names appear in alphabetic order along with the numbers of the lines on which they are referenced or defined. Line numbers on which the symbol is defined are flagged with '#'.

13 Output to Display or Printer

To make the listing file go to the display or printer rather than to disk, use the following format:

*,output device=source file

where output device is TTY (display) or LPT (printer).

Examples:

*,TTY=TEST	Assemble file TEST/MAC and output a listing file to the display.
*,LPT=TEST	Assemble file TEST/MAC and output a listing file to the printer.
*TEST,TTY=TEST	Assemble file TEST/MAC; create a disk file, TEST/REL; output a listing file to the display.

INDEX

ABS 188, 19Ø, 237
accessing subroutines 139
A command 28, 36, 228
A field 1ØØ
AINT 188, 2ØØ, 237
ALEDIT 23, 37
ALEDIT Command Mode Keys, Table 1 27
  <left arrow> 27
  <down arrow> 27
  <up arrow> 27
  <CTRL><A> 27
  <CTRL><B> 27
  <.> 27
  #line<ENTER> 27
  <BREAK> 27
  <SHIFT> 27
  <up arrow> 27
ALEDIT Editor Commands, Table 2 28
  current line 28
  del 28
  string 28
  text 28
  A 28
  B 28
  C del string1 del string2
    del occurrence 28
  D 29
  E 29
  F del string del occurrence 3Ø
  G 3Ø
  H 3Ø
  I 31
  J 31
  K 31
  L filespec $C 31
  L TEST $C 32
  M 32
  N 32
  O 33
  P 33

Q 33
R 33
T 33
U 33
V 33
W filespec $option1... 33
options 34
E 34
L, ML, or LM 34
M 34
ALEDIT Line Edit Mode Special Keys 37
<SPACEBAR> 37
<SHIFT><up arrow> 37
<right arrow> 37
<left arrow> 37
<ENTER> 37
ALEDIT Line Edit Mode Subcommands 36
A 36
E 36
Hstring 36
Istring 36
L 36
nCstring 36
nDstring 36
nKcharacter 36
nScharacter 37
Q 37
Xstring 37
ALOG 188, 201, 237
ALOG10 188, 201, 237
AMAX0 188, 203, 237
AMAX1 188, 203, 237
AMIN0 188, 205, 237
AMIN1 188, 205, 237
AMOD 188, 206, 237
.AND. 92
ANSI 59, 247
arrays 143
declarators 65
declarator statements 69
variables 81-82
arithmetic

expressions 87-88
hierarchy 93-94
IF 163
operators 87
ASCII character codes 257
ASSIGN 64, 136
assigned GO TO 162
assignment statements 83
ATAN 188, 191, 237
ATAN2 188, 192, 237
B command 28
BLOCK DATA 66, 128, 137, 237
Boolean logic 92
buffers 95
BYTE 65, 76, 138, 166, 237
byte data 75
CALL 63, 125, 127, 135, 139, 237
CALL OPEN 96, 171
CALL OUT 238
CALL POKE 174, 238
calling convention 47
carriage controls 98
carriage control character 155
carriage control statements 112-13
C command 28
C del string1 del string2 del occurrence 28
/CMD 19
coding sheet 62
command file 19, 50
Command Mode 24-26
T 25
1 26
2 26
O 26
D 26
common logarithm 202
COMMON
DATA storage 85
statements 69, 238
storage 129
variable storage 85
compiler 13, 22, 39-49

commands 234
errors 217-2Ø
runtime error message 2Ø
switches 41-49, 234
-H 42,234
-M 42, 46, 235
-N 21, 42, 44, 53, 234
-O 42
-P 46, 51, 54, 235-36
complex expression 87
complex logical expressions 92
computed GO TO 161
constants 75, 87
continuation line 72
continuation marker 61
CONTINUE 64, 68, 141, 147, 238
control statement 63, 72
COS 122, 188, 193, 238
creating FUNCTIONS 188-89
current line command 28
cursor position 25-28, 33, 36-37
DABS 188, 19Ø, 238
data 75-86
DATA 142, 238
data files 95-119
data initialization statements 66, 69
DATAN 188, 191, 238
DATAN2 188, 192, 238
DATA statements 46, 84
data types 75
DBLE 188, 238, 294
D command 29
DCOS 188, 238, 293
declaration statement 14Ø
DECODE 64, 143-44, 239
default 54, 166
definition statements 66-67, 69
del (delete) command 28
delimiters 98, 117
DEXP 196, 239
D field 1Ø1

DIM 188, 195, 239
DIMENSION 65, 145, 152, 239
direct addressing 118-19
disk memory editor 24
DLOG 188, 201-02, 239
DLOG10 239
DMAX1 188, 203, 239
DMIN1 188, 205, 239
DMOD 188, 206, 239
DO 63, 146-48, 240
DO loops 68, 178
double precision
  data 75-76, 103-104
  numbers 78
  statement 63, 101, 148
-D switch 54, 236
DSIGN 188, 208, 240
DSIN 188, 209, 240
DSQRT 188, 211, 240
E command 25, 29, 34
E field 102
editor 11-13, 23-38
editor errors 222-23
-E switch 19, 51, 236
ENCODE 64, 157, 240
END 64, 158, 240
END= 115-16, 177
END statement 69
ENDFILE 64, 159, 240
ENDFILE command 97
<ENTER> special key 37
.EQ. 90
EQUIVALENCE 66, 69, 152-53, 240
EQUIVALENCE statement 85, 132-33
ERR= 115-16, 177, 185
evaluation hierarchy 88
executable statements 63-64, 69, 135, 164
EXP 188, 196, 240
exponential notation 77
expressions 84, 87-94
extended integer data 75
extended integers 76

extended integer variable 169
extension 15
EXTERNAL 154, 240
EXTERNAL statements 69
F80 13, 39, 125
F80 commands 39
.FALSE. 79
F command 30
F FORMAT field 102-03
field
  descriptors 99, 155-56
  fields 95
  separators 156
  specification 70
filename 12
files 95-119
filespec 23, 30
FLOAT 188, 198, 240
floating point 70
/FOR 14
FORLIB/REL 17, 95
format
  data 79
  I/O 155
  label 115
  specification 70, 112
  statement 70-71, 155
FORTRAN
  character set 61
  functions 87-88, 187-212
  source programs 23
  statements 12, 61, 65, 135-86
FUNCTION 67, 122-23, 158, 241
FORLIB Arithmetic Library Subroutines 258-60
G command 30
G descriptor 104
G fields 105
.GE. 90
globals 20, 49
GOTO 63, 67, 160-62, 241
.GT. 90
H command 30

H descriptor 1Ø5
H field 1Ø5
hexadecimal data 75
hexadecimal notation 17
hexadecimal numbers 8Ø
H FORMAT descriptor 1Ø6
Hstring 36
-H switch 42, 234
hyperbolic tangent 212
IABS 188, 195, 241
I command 24-25, 31
identification field 61-62
IDIM 188, 195, 241
IDINT 188, 2ØØ, 241
IF 63, 163-64, 241
I fields 1Ø6
IFIX 188, 197, 241
IMPLICIT 65, 166, 241
IMPLICIT declaration 81
INCLUDE 63, 167, 241
$INIT 16, 45
initialize routine 46
INP 64, 188, 199, 241
input/output 45
  device 95
  fields 97
  interface 249
  ports 172
  statements 63, 7Ø
Insert mode 24
INT 188, 2ØØ, 241
integer data 75
integers 1Ø6
INTEGER 64-65, 75-76, 166, 168, 242, 262
INTEGER*4 65, 166, 169, 263
intra-program branching 66-67
ISIGN 188, 2Ø8, 242
I specifications 1Ø6
J command 31
K command 31, 36
L8Ø 18, 49
L command 31, 36, 231

Language Extensions and Restrictions 247
L descriptor 107
.LE. 90
L fields 107
library function 122
line edit mode 24, 35
LINK
  LINK-80 18
  link-compatible object files 264
  linker 18, 39, 49-56
    commands 50, 236
    errors 224-26
    switches 51-56, 236
      -D 54, 236
      -E 19, 51, 236
      -N 21, 42, 44, 53, 234
      -P 46, 51, 54, 235-36
      -R 51, 55, 236
      -S 51, 55, 236
      -U 55, 236
listing addresses 44
listing file 16, 41, 45
literal
  data 75
  expression 94
  literals 79
logical
  data 75
  expressions 87
  IF 91
  logical units 95
  logical unit number 95
  operands 92
  operators 92
  statement 65, 79, 107, 166, 170, 242, 262
looping 146
LM 34
.LT. 90
L TEST 31-32
LUN 95-96, 112, 116, 151, 171, 177, 189
$LUNTB 47, 248
$MAIN 13, 131

MAXØ 188, 2Ø3, 242
MAX1 188, 2Ø3, 242
M command 32, 34
memory location 16, 152
MINØ 188, 2Ø5, 242
MIN1 188, 2Ø5, 242
ML 34
MOD 188, 2Ø6, 242
-M switch 42, 46, 235
natural logarithm 2Ø1
N command 32, 231
nCstring 36
nDstring 36
.NE. 9Ø
negative indicators 87
nKcharacter 36
nScharacter 37
-N switch 21, 42, 44, 53, 234
non-executable statements 64, 135
.NOT. 93
object file 18
occurrence 28
O command 33, 231
OPEN 135, 171, 242
OPEN SUBROUTINE 96
operands 87
operators 87
.OR. 92
originating address 17
-O switch 42
OUT 64, 135, 172, 242
output fields 98
output file format 261
parameter blocks 46
parameters 124-33, 253-54
parentheses 156
PAUSE 63, 173, 242
pointer 182
POKE 25, 174, 242
positive indicators 87
print command (H) 3Ø
PROGRAM 66, 175, 243

program execution 18
program branching 66
pseudo-op 63
-P switch 46, 51, 54, 235-36
Q command 33, 37, 231
Quit command 13, 33, 231
RAN 176, 243
R command 33, 231
READ 64, 7Ø, 95, 112-15, 177-79, 243
REAL 65, 17Ø, 18Ø, 243, 262
real data 75, 1Ø3-Ø4
real number 77
record length 96
REC= 112, 115, 177, 185
/REL 17
relational expressions 9Ø
relocatable
  addresses 14, 17
  data address 17
  object code 14, 39
  object file 14, 22, 4Ø, 264
  program address 17
replacement line 7Ø
replacement statements 62-63
RETURN 63, 181, 243
REWIND 182, 243
ROM 42, 46
-R switch 51, 55, 236
S command 37
scalar variables 81-82
scaling factor 1Ø8
search 37
segmenting programs 121-33
sequential addressing 117-19
SIGN 188, 2Ø9, 243
skipping spaces 1Ø9
SNGL 188, 21Ø, 243
source filespec 23-24
source program 14
specification list 11Ø, 157
SQRT 122, 188, 243
-S switch 51, 55, 236

stack space 42
statement labels 61-62
statement order 69
STOP 63, 183, 243
storage
  definers 66
  definer statements 69
  format 75, 262-63
string1 28
string2 28
strings stored in variables 1ØØ
subprogram branching 66-67
subprogram linkages 254
SUBROUTINE 17, 66, 123-27, 184, 243
subroutine library requests 2Ø
subroutine requests 19
subscript 82-83
system library 55
TANH 188, 212, 244
T command 33, 231
termination 15Ø
text command 28
trigonometric functions 122
TRSDOS command file 11
TRSDOS errors 215-16
.TRUE. 79
type specification statement 64, 69
U command 33
unconditional GO TO 16Ø
undefined globals 2Ø, 49, 55
unformatted data 98
-U switch 55, 236
variable 75, 87
variable list 115
V command 33
W command 33, 232
WRITE 63-64, 68-69, 95, 115-16, 188, 244
W SAMPLE 34
W TEST 26
X command 35
X field 1Ø9
.XOR. 92

Xstring 37
Z-8Ø assembly language 17, 234
Z command 32

IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE LICENSED ON AN "AS IS" BASIS WITHOUT WARRANTY.

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

RADIO SHACK SOFTWARE LICENSE

A. Radio Shack grants to CUSTOMER a non-exclusive, paid up license to use on CUSTOMER'S computer the Radio Shack computer software received. Title to the media on which the software is recorded (cassette and/or disk) or stored (ROM) is transferred to the CUSTOMER, but not title to the software.

B. In consideration for this license, CUSTOMER shall not reproduce copies of Radio Shack software except to reproduce the number of copies required for use on CUSTOMER'S computer (if the software allows a backup copy to be made), and shall include Radio Shack's copyright notice on all copies of software reproduced in whole or in part.

C. CUSTOMER may resell Radio Shack's system and applications software (modified or not, in whole or in part), provided CUSTOMER has purchased one copy of the software for each one resold. The provisions of this software License (paragraphs A, B, and C) shall also be applicable to third parties purchasing such software from CUSTOMER.

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U.K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.